TURING 图灵程序设计丛书

Apress®

Pro C# 5.0 and the .NET 4.5 Framework

Sixth Edition

精通C#（第6版）

Jolt大奖提名图书，世界级C#专家之作，难以超越的畅销经典

[美] Andrew Troelsen 著

姚琪琳 朱晔 肖逵 张大磊 王少葵 范睿 等 译



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Pro C# 5.0 and the .NET 4.5 Framework
Sixth Edition

精通C#（第6版）

[美] Andrew Troelsen 著

姚琪琳 朱晔 肖逵 张大磊 王少葵 范睿 等 译

人 民 邮 电 出 版 社

北 京

图书在版编目（C I P）数据

精通C# : 第6版 / (美) 特罗尔森 (Troelsen, A.)
著 ; 姚琪琳等译. -- 北京 : 人民邮电出版社, 2013.7
(图灵程序设计丛书)
书名原文: Pro C# 5.0 and the .NET 4.5
framework, sixth edition
ISBN 978-7-115-32181-7

Ⅰ. ①精… Ⅱ. ①特… ②姚… Ⅲ. ①C语言—程序设
计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第133321号

内 容 提 要

本书是 C# 领域久负盛名的经典著作，深入全面地讲解了 C# 编程语言和 .NET 平台的核心内容，并结合大量示例剖析相关概念。全书分为八部分：C# 和 .NET 平台、C# 核心编程结构、C# 面向对象编程、高级 C# 编程结构、用 .NET 程序集编程、.NET 基础类库、WPF 和 ASP.NET Web Forms。第 6 版是对第 5 版的进一步更新和完善，内容涵盖了最先进的 .NET 编程技术和技巧，并准确呈现出 C# 编程语言的最新变化和 .NET 4.5 Framework 的新特性。

本书由微软 C# MVP Andrew Troelsen 编写，第 6 版专门针对 C# 5.0 和 .NET 4.5 进行了细致入微的修订，是各层次 .NET 开发人员的必读之作。

◆ 著　[美] Andrew Troelsen
译　姚琪琳　朱　晔　肖　逵　张大磊
王少葵　范　睿 等
责任编辑　刘美英
执行编辑　刘美英　李　洁
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：76.75
字数：1813千字　2013年7月第1版
印数：1－4 000册　2013年7月北京第1次印刷

著作权合同登记号　图字：01-2012-7085号

定价：159.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

广告经营许可证：京崇工商广字第 0021 号

前　言

大约 2001 年，我有幸获得一个机会，为即将兴起的一种微软技术撰写一本书，那时这种技术被称为 NGWS（Next Generation Windows Software，下一代视窗服务）。在检查微软提供的源代码的过程中，我发现大量代码注释与一种 COOL（Common Object Oriented Language）编程语言相关。

在我使用预览版创作 *C# and the .NET Platform*（第 1 版）初稿的过程中，NGWS 最终更名为微软.NET 平台，而 COOL，你应该能猜到，就是今天众所周知的 C#。

本书第 1 版与.NET 1.0 Beta 2 同时推出。从那以后，随着 C#编程语言的更新以及新发布的.NET 平台引入大量 API，本书内容也不断修订。

这些年来，本书得到了出版社（曾入围 Jolt 大奖提名，获得 2003 年 Referenceware 编程类图书卓越大奖）、读者、计算机科学和软件工程类大学课程的认可。对此我甚是感激。

更重要的是，很高兴收到全世界读者和教育工作者发来的邮件，与他们交谈感觉很棒。感谢所有人的意见、评论，当然还有批评。也许有些邮件我来不及回复，但我充分考虑了每一封邮件提出的问题，这一点可以保证。

你和我，我们是一个团队

技术作家所面对的是一群苛刻的读者（我知道，因为我就是他们中的一员）。无论使用什么平台，对部门、公司、客户和任何课题来说，构建软件解决方案都是非常复杂而且有针对性的事情。可能你在电子出版行业工作，或者为政府开发系统，或者是在科研机构或军队的某个部门工作。就我自己而言，我开发过儿童教育软件（Oregon Trail、Amazon Trail 等游戏软件）、各种 *n* 层系统以及许多医疗和金融行业的项目。你工作时编写的代码和我编写的代码百分之百是不同的（除非我们恰巧以前在一起工作）。

因此，在这本书中，我特意避免选择那些和具体行业紧密相关的例子，而是用与行业无关的例子来解释 C#、OOP、CLR 和.NET 基础类库。我不使用诸如数据填充表格、薪水计算或者其他的一些例子，而是坚持用与我们都有联系的主题：汽车，另外再加上几何结构和雇员薪水系统作为补充示例。你不用担心会有什么陌生的背景知识。

我要做的是尽最大可能解释 C#编程语言和.NET 平台的核心内容。同时，我会尽可能把进一步学习本书的工具和策略提供给你。

你要做的是理解这些内容并将其付诸具体编程工作中。我很清楚，你的项目可能与具有友好昵称的汽车（比如，BMW 的 Zippy 和 Yugo 的 Clunker）根本无关，但是所用到的知识是相通的。

放心，只要理解了这本书中的概念，你便能够很好地构建一个和实际编程环境紧密相关的.NET 解决方案。

本书内容

本书从逻辑上分为 8 个部分，每个部分包含一些相关联的章节。下面先按部分，再按章来分解本书。

第一部分：C#与.NET平台

第一部分的目的在于让你初步适应并了解.NET 平台以及在构造.NET 应用中用到的各种开发工具（其中很多是开源的）。

第 1 章：.NET 之道

这一章讲述本书其余部分的脉络。该章的主要目的是介绍许多.NET 相关的构建块，如 CLR（公共语言运行库）、CTS（公共类型系统）、CLS（公共语言规范）以及基础类库。该章让你对 C#编程语言和.NET 程序集格式有一个初步了解，此外，本章还会介绍 Windows 8 操作系统中.NET 平台的作用，并讲述 Windows 8 应用程序和.NET 应用程序之间的区别。

第 2 章：构建 C#应用程序

这一章介绍如何使用各种工具和技术来编译 C#源代码文件。先介绍了如何使用命令行编译器（csc.exe）和 C#响应文件，接着介绍了许多代码编辑器和 IDE（集成开发环境），包括 Notepad++、SharpDevelop、Visual C# Express 以及 Visual Studio；同时也会介绍如何通过在本地安装.NET Framework 4.5 SDK 文档来配置自己开发用的电脑。

第二部分：C#核心编程结构

这部分很重要，因为所有类型的.NET 软件开发都必须用到它，如 Web 应用、GUI 桌面应用、代码库和 Windows 服务。你将会在这里了解.NET 基本数据类型，学习文本处理以及各种 C#参数修饰符的作用（包括可选参数和命名参数）。

第 3 章：C#核心编程结构 I

这一章正式开始研究 C#编程语言，其中介绍了 `Main()`方法的作用和.NET 平台的内部数据类型，以及使用了 `System.String` 和 `System.Text.StringBuilder` 的文本数据操作。本章还讨论了迭代和选择构造、宽化和窄化操作以及 `unchecked` 关键字的用法。

第 4 章：C#核心编程结构 II

这一章完成了 C#核心方面的研究，首先介绍重载类型方法的结构以及如何通过 `out`、`ref` 和 `params` 关键字定义参数，接着介绍了 C#的两个特性——参数（argument）和可选参数（optional parameter），然后介绍了如何创建和操作数据数组，定义可空数据类型（使用?和??操作符）以及值类型（包括枚举和自定义结构）和引用类型之间的区别。

第三部分：C# 面向对象编程

从这一部分开始我们深入学习 C#语言的核心构造，其中包括面向对象编程语言（OOP）的细节。此外，我们还要研究一下如何处理运行时异常，并详细阐述强类型接口的使用方法。

第 5 章：封装

从这一章开始，我们会研究如何使用 C#编程语言进行面向对象编程。在探讨了 OOP 的支柱概念（封装、继承和多态）之后，会介绍如何使用构造函数、属性、静态成员、常量以及只读字段来构建健壮的类类型。我们最后会研究分部类型定义、初始化对象的语法和自动属性。

第 6 章：继承和多态

这一章研究 OOP 其余的支柱概念（继承和多态），通过它，我们就能构建相关类类型的家族。我们还会研究虚方法、抽象方法（和抽象基类）以及*多态接口*的本质。这一章最后介绍.NET 平台最高层基类 `System.Object` 的作用。

第 7 章：结构化异常处理

这一章的关键在于讨论如何使用结构化异常处理来处理运行时的异常情况。你不但可以学到 C#控制这些异常的关键字（`try`、`catch`、`throw` 和 `finally`），还将了解应用程序级异常和系统级异常的区别。另外，该章还讲述了 Visual Studio 中不同的调试工具，这些工具能让你调试那些被忽略的异常。

第 8 章：接口

这一章的内容建立在对面向对象开发的理解之上，涵盖了基于接口的编程主题。你将学到如何定义类和支持多行为的结构，如何在运行时发现这些行为，以及如何使用*显式接口*实现来选择性地隐藏特定的行为。除了创建大量自定义接口外，还介绍了如何实现.NET 平台中的标准接口，以及使用这些接口构建可以排序、复制、枚举和比较的对象。

第四部分：高级C#编程结构

这部分介绍了许多重要的高级技术，能让你对 C#语言有更深的理解。通过对接口和委托的学习，你将会了解.NET 类型系统；同时你会学到泛型的作用，初步了解 LINQ，以及更多 C#高级特性（例如扩展方法、分部方法和指针操作）。

第 9 章：集合与泛型

这一章首先研究泛型（generic）的概念。你将看到，泛型编程提供了一种创建类型和类型成员的方式，它包含由调用者指定的变量占位符。简而言之，泛型编程大大加强了应用程序的性能和类型安全。你不仅可以在 `System.Collections.Generic` 命名空间中看到各种泛型，而且可以学习如何创建自己的泛型方法和类型（有限制或没有限制）。

第 10 章：委托、事件和 Lambda 表达式

这一章的目的在于阐明委托（delegate）类型。可以简单地认为，一个.NET 委托就是指向应用程序中其他方法的一个对象。使用这个类型，可以构建允许多个对象进行双向会话的系统。在分析了.NET 委托的使用后，将介绍 C#的 event 关键字，使用这个关键字可以简化原始委托编程的操作。随后研究 C# Lambda 操作符`=>`，并探讨了委托、匿名方法和 Lambda 表达式之间的联系。

第 11 章：高级C#语言特性

这一章介绍了许多高级的.NET 编程技术，这些技术能够让你对 C#编程语言有更深的理解。例如，你将学到如何重载操作符、创建自定义类型转换例程（包括隐式的和显式的）、构建类型索引器并与之交互，使用扩展方法、匿名类型、分部方法以及使用不安全的代码上下文来操作 C#指针。

第 12 章：LINQ to Object

这一章开始研究 LINQ（语言集成查询），它可以用来构建强类型的查询表达式，并且可以把它应用到很多 LINQ 目标来操作最广义的数据。我们还会学习把 LINQ 表达式应用到数据容器（例如，数组、集合和自定义类型）的 LINQ to Object。这对于本书其余部分所涉及的其他 LINQ API 同样适用（如 LINQ to XML、LINQ to DataSet、PLINQ 和 LINQ to Entity）。

第 13 章：对象的生命周期

这一章分析了 CLR 如何使用.NET 垃圾收集器来管理内存。你将会了解应用程序根、对象产生和 System.GC 类型的作用。理解了这些之后，接下来学习可处置的对象（通过 IDisposable 接口）和终结过程（通过 System.Object.Finalize()方法）。这一章还将研究 Lazy<T>类，它用来定义一个直到调用者发出请求时才会分配的数据。你将看到，当你的程序不希望托管堆散乱不堪时，这个特性将有助于确保堆的整洁。

第五部分：用.NET程序集编程

这一部分深入分析了.NET 程序集格式的细节。你不仅会学到如何部署和配置.NET 代码库，而且会理解.NET 二进制映像的内部结构。这部分也阐述了.NET 特性的作用和运行时解析类型信息的作用，还介绍了动态语言运行时（DLR, Dynamic Language Runtime）和 C# dynamic 关键字的作用。后面的章节分析了一些程序集相关的高级主题（如应用程序域、CIL 语法和在内存中构建程序集）。

第 14 章：.NET 程序集入门

从一个比较高的层次来看，程序集（assembly）是用于描述托管的*.dll或*.exe二进制文件的一个术语，但是其真正内涵实际上远远不仅于此。通过这一章，你将学到单文件程序集和多文件程序集的区别，以及如何构建和部署每一个实体；你还会学到如何利用基于XML的*.config文件和发布策略程序集来配置私有的和共享的程序集。通过这些内容，你将看到GAC（全局程序集缓存）的内部结构。

第 15 章：类型反射、晚期绑定和基于特性的编程

这一章通过 System.Reflection 命名空间，分析了运行时类型发现的过程，来继续探讨.NET 程序集。使用这些类型，你能够创建一个可以实时读取程序集元数据的应用程序。该章还将介绍如何在运行时使用晚期绑定来动态加载和创建类型。最后的主题是.NET 特性（包括标准的和自定义的）的作用。为了说明每个主题的用法，该章最后将构造一个可扩展的 Windows Forms 应用程序。

第 16 章：动态类型和动态语言运行时

.NET 4.0 引入了一个新的.NET 运行时环境，叫做动态语言运行时。使用 DLR 和 C# 2010 dynamic 关键字，可以定义直到运行时才真正处理的数据。这些特性可以显著地简化一些极其复杂的.NET 编程任务。在这一章中，你将学习一些动态数据的实际用法，包括如何用简单的方式使用.NET 反射 API，以及如何用最小的代价与遗留的 COM 库通信。

第 17 章：进程、应用程序域和对象上下文

既然你已经对程序集有了一定的了解，这一章将深入探讨加载的.NET 执行单元的组成。这一章的目标是阐明进程、应用程序域和上下文边界的关系。该章所叙述的内容为第 19 章做了很好的铺垫，那一章讨论的是多线程应用程序的构造。

第 18 章：CIL 和动态程序集的作用

这一章有两个目的。在前半部分（大致）中，将会比之前章节更具体地介绍 CIL 的语法和语义，余下的部分主要讲述 `System.Reflection.Emit` 命名空间的作用。使用这些类型，可以构建一个能在运行时在内存中产生.NET 程序集的软件。正式地说，一个能在内存中定义并执行的程序集称为动态程序集。

第六部分：.NET基础类库

到本书的这一部分，你应该已经很好地掌握了 C#语言以及.NET 程序集格式的细节。这一部分将通过探索基础类库中的一些常用服务程序来讲授一些新的知识，包括多线程应用程序的创建、文件的输入/输出和利用 ADO.NET 的数据库访问，通过 WCF 构造分布式应用程序以及构建使用了 WF API 和 LINQ to XML API 的支持工作流的应用程序。

第 19 章：多线程、并行和异步编程

这一章介绍了如何构建多线程应用程序，演示了大量用于编写线程安全代码的技术。首先复习了.NET 委托类型，解释了委托对于异步方法调用的内在支持。接下来，研究了 `System.Threading` 命名空间中的类型。最后，介绍了 TPL（Task Parallel Library，任务并行库）。使用 TPL，.NET 开发者可以用一种极其简单的方式,将应用程序的工作分配给所有可用的 CPU。与此同时,你还将学习 PLINQ 的作用，它提供了一种在多个机器内核中执行 LINQ 查询的方法。本章结尾处总结了.NET 4.5 引入的几个新的 C#关键字，它们能够将异步方法调用直接集成到语言中。

第 20 章：文件输入输出和对象序列化

`System.IO` 命名空间允许与机器的文件和目录结构交互。通过这一章的学习，你将学会如何以编程方式创建（和删除）一个目录系统，以及如何将数据从不同的（例如，基于文件的、基于字符串的、基于内存的等）数据流中移进移出。本章的后半部分探讨了.NET 平台的对象序列化服务。简单地说，序列化（serialization）就是将一个对象（或一组相关对象）的状态持久化为流，以便今后使用。反序列化（Deserialization）是一个从流中取出对象并放入供应用程序使用的内存中的过程。只要理解了基本原理，你就会学到如何通过 `ISerializable` 接口和一组.NET 特性来定制序列化过程。

第 21 章：ADO.NET 之一：连接层

本书中有 3 章介绍数据库相关操作，这是第一章，介绍.NET 平台 ADO.NET 的数据库访问 API。具体而言，涉及.NET 数据提供程序的作用以及如何使用由连接对象、命令对象、事务对象以及数据读取器对象构成的 ADO.NET 连接层与关系数据库进行通信。该章引导你创建一个在本书剩余部分会用到的自定义数据库和自定义数据访问类库（AutoLotDAL.dll）。

第 22 章：ADO.NET 之二：断开连接层

这一章将通过研究 ADO.NET 的*断开连接层*继续探讨数据库操作。在这里，我们会学习 `DataSet` 类型、数据适配器对象以及各种可以大幅简化创建数据驱动应用程序的 Visual Studio 工具。此后，我们会学习如何把 `DataTable` 对象绑定到用户界面元素以及如何使用 LINQ to DataSet 将 LINQ 查询应用于内存对象 `DataSet`。

第 23 章：ADO.NET 之三：Entity Framework

这一章研究了 ADO.NET 的最后一部分内容——Entity Framework（EF）的作用。实质上，EF 是

一种用直接映射到业务模型上的强类型类编写数据访问代码的方式。在这里，你将了解 EF 对象服务的作用、实体客户端和对象上下文，以及*.edmx 文件的构成。同时，还将学习如何使用 LINQ to Entity 与关系型数据库交互。你还将构建最终版本的自定义数据访问库（AutoLotDAL.dll），本书其他章节中将会使用这个库。

第 24 章：LINQ to XML 简介

第 14 章介绍核心的 LINQ 编程模型，特别是 LINQ to Object。在这里，我们会通过研究如何把 LINQ 查询应用到 XML 文档来深入理解 LINQ。我们首先学习在使用 System.XML.dll 程序集的类型进行.NET XML 操作时所暴露出的一些“缺点”，然后探索如何使用 LINQ 编程模型（LINQ to XML）在内存中创建 XML 文档、将文档持久化到硬盘驱动以及对其内容进行导航。

第 25 章：WCF

到现在为止，所有的示例程序都是在一台计算机上执行的。在这一章中，我们将学习WCF API以便无须考虑底层管道，以系统的方式构建分布式应用程序。该章会介绍WCF服务、服务器端以及客户端的构建。我们还会看到，WCF服务非常灵活，在客户端和服务器端中都可以使用基于XML的配置文件以声明方式指定地址、绑定和契约。

第 26 章：Windows Workflow Foundation 简介

本章你首先会学习启用工作流的应用程序的作用，随后了解使用.NET 4.0 WF API 对业务流程进行建模的不同方式，接下来学习 WF 活动库的范围，以及如何构建自定义活动。在这个活动中，将使用本书前面所创建的自定义数据库访问库。

第七部分：WPF

.NET 3.0 向程序员介绍了一个绝妙的 API——WPF（Windows Presentation Foundation），该 API 很快就成为 Windows Forms 桌面编程模型的继承者。实质上，WPF 所构建的桌面应用程序可以包含向量图形、交互式动画，以及使用声明式标记语法 XAML 所进行的数据绑定操作。

此外，WPF 控件架构提供了一种非常简单的方式，仅仅使用一些格式良好的 XAML，就可以彻底改变常用控件的外观。

第 27 章：WPF 和 XAML

本质上，WPF 可以为桌面应用程序（以及非直接的 Web 应用程序）构建交互性极好的富媒体前端。和 Windows Forms 不同，这个超强的 UI 框架把很多关键服务（例如 2D 和 3D 图形、动画、富文档等）整合到了一个统一的对象模型中。在这一章中，我们会从 WPF 以及 XAML（可扩展应用程序标记语言）开始研究，还会学习如何创建不使用 XAML、只使用 XAML 以及两者结合的 WPF 程序。最后，我们创建了在其余 WPF 相关章节中都会用到的自定义 XAML 编辑器。

第 28 章：使用 WPF 控件编程

这一章将介绍使用内置的 WPF 控件和布局管理器的过程，例如，构建菜单系统、拆分窗口、工具条和状态栏，还将介绍大量 WPF API（及其相关控件），包括 WPF Documents API、WPF Ink API 和数据绑定模型。同样重要的是，这一章将开始研究 Expression Blend IDE，它将简化为 WPF 应用程序创建富 UI 的任务。

第 29 章：WPF 图形呈现服务

WPF 是一个图形密集型 API，它提供了 3 种呈现图形的方式：形状、绘图和几何图形、可视化。在这一章中，我们将介绍这几种方法，并讨论大量重要的图形基元（如画刷、画笔和图形变换），还将学习 Expression Blend 用于简化创建 WPF 图形过程的多种方式，以及如何对图形数据执行命中测试操作。

第 30 章：WPF 资源、动画和样式

这一章介绍了 3 个重要的（也是相互关联的）主题，它们将加深你对 WPF API 的理解。第一个主题是逻辑资源的作用。你将看到逻辑资源（也叫对象资源）系统提供了一种方式，可以在 WPF 应用程序中命名和引用常用的对象。接下来，你将学习如何定义、执行和控制动画序列。你可能会认为 WPF 动画仅局限于视频游戏和多媒体应用，但实际上绝不是这样。最后你将学习 WPF 样式的作用。与使用 CSS 或 ASP.NET 主题引擎的 Web 页面类似，WPF 应用程序也可以定义常用控件的外观。

第 31 章：依赖属性、路由事件和模板

本章先介绍了创建自定义控件时涉及的两个重要话题：依赖属性和路由事件。理解了这些内容之后，我们将学习默认模板的作用，以及如何在运行时以编程的方式查看它们。打好这些基础之后，最后将学习如何创建自定义模板。

第八部分：ASP.NET Web Form

这一部分主要研究使用 ASP.NET 编程 API 来构建 Web 应用程序。我们会看到，ASP.NET 基于标准的 HTTP 请求/响应对事件驱动的面向对象框架分层，以此来对桌面用户界面的创建进行建模。

第 32 章： ASP.NET Web Form

本章开始介绍使用 ASP.NET 进行 Web 应用开发。如你所见，服务器端的脚本代码现在由“真正的”面向对象的语言（如 C#和 VB .NET 等）所替代。这一章将介绍 ASP.NET 网页的构造、基础的编程模型以及 ASP.NET 的其他关键主题，如 Web 服务器的选择和 web.config 文件的使用。

第 33 章：ASP.NET Web 控件、母版页和主题

由于前几章介绍了 ASP.NET 页面对象的构建，这一章将会关注组成内部控件树的控件。在这里，我们会研究包括验证控件、内置站点导航控件以及各种数据绑定操作在内的核心 ASP.NET Web 控件。同样，还会演示母版页的作用以及与传统样式表对应的服务器端 ASP.NET 主题引擎。

第 34 章：ASP.NET 状态管理技术

这一章将通过研究.NET 下处理状态管理的多种方式来扩展你对 ASP.NET 的理解。和传统的 ASP 一样，使用 ASP.NET 可以轻松创建 cookie 以及应用程序级变量和会话级变量。然而，ASP.NET 还引入了一项新的状态管理技术：应用程序高速缓存。只要知道了使用 ASP.NET 处理状态的多种方式，你就会明白 `HttpApplication` 基类的作用，以及如何用 web.config 文件动态改变 Web 应用程序的运行时行为。

附录

除了 34 章正文，本书英文版还包括附录 A 和附录 B，其内容需要从 Apress 网站（www.apress.com）的本书主页下载。附录 A 涵盖基本 Windows Forms API，正文中有几个 UI 示例用到了。附录 B 通过

Mono 平台研究了.NET 不依赖于平台的特性。

附录 A：Windows Forms 编程

.NET 平台最初发布的桌面 GUI 工具包是 Windows Forms。该附录将介绍这个 UI 框架的作用，并演示如何构建主窗口、对话框和菜单系统，还将说明窗体继承的作用，以及如何使用 `System.Drawing` 命名空间呈现二维图形数据。最后，该附录将构建一个绘图程序（半成品），演示附录中讨论的各个主题。

附录 B：使用 Mono 进行平台无关的.NET 开发

最后，附录 B 介绍了一个叫做 Mono 的.NET 平台的开源实现。使用 Mono 可以在 Mac OS X、Solaris 以及各种 Linux 版本的操作系统中创建、部署和执行富特性的.NET 应用程序。由于 Mono 和微软的.NET 平台非常相似，因此你应该知道 Mono 提供的大部分内容。因此，附录 B 会重点介绍 Mono 的安装过程、Mono 的开发工具以及 Mono 运行时引擎。

本书源代码

本书所包含的所有代码示例都可以从 Apress 网站上的 Source Code/Download 中免费下载（也可以从图灵社区本书主页 http://www.ituring.com.cn/book/1046 免费下载）。访问网址 http://www.apress.com，选择 Source Code/Download 链接，然后按书名查找。找到本书的主页后，就可以下载一个压缩的*.zip 文件。解开压缩文件就会看到，所有代码都是按章编排的。

需要提醒你注意的是，本书的很多章节都包含有如下所示的源代码说明，书中讨论的例子都可以依此线索下载，并加载到 Visual Studio 中，以便进一步讨论和修改。

源代码　这里给出了源代码在压缩文件中的具体目录。

要打开一个 Visual Studio 解决方案，可以使用 File→Open→Project/Solution 菜单选项，然后导航到解压缩文档所在的子目录，找到正确的*.sln 文件。

勘误信息

阅读本书过程，你或许偶尔会发现一些语法错误或代码错误（很显然我不希望看到这些）。如果真发现了，我在此道歉。作为一个凡人，尽管我已经很尽力了，但是一两个小错误总是难免的。你可以从 Apress 的网页上获得勘误表（还是在这本书的“主页”上）。如果你发现错误的话，请在那里找到我的联系方式，与我联系。

联系作者

如果你有任何关于本书源代码的问题，或者需要进一步阐明这里所举的例子，亦或者只是想简单地向我传达你关于.NET 平台的想法，请通过以下电子邮件地址与我联系：atroelsen@intertech.com（为了确保你的邮件不会被我的信箱划为垃圾邮件，请在主题栏中包含“C# SixthEd”）。

请你们相信，我会尽我所能在较短的时间里回复你们。但是，就如同各位一样，我有时也会比较忙。如果我没能在一周或两周的时间里回复你们，请不要认为我是一个怪异的人或者不屑与你们交流，我可能只是比较忙而已（或者，如果足够幸运的话，在某处度假也不一定）。最后，感谢购买本书（或者至少在你决定是否购买的时候，曾经在书店里翻看过）。我希望你喜欢阅读本书，并且可以灵活运用书中所学的知识。

Andrew Troelsen

目　录

第一部分　C#与.NET平台

第二部分 C#核心编程结构

第三部分 C#面向对象编程

第四部分　高级C#编程结构

第五部分 用.NET程序集编程

第六部分　.NET基础类库

第七部分 WPF

Part 1

第一部分

C# 与 .NET 平台

本部分内容

第1章

.NET之道

微软的.NET平台（以及相关的C#编程语言）自2002年正式发布后很快成为软件开发的主流平台。本书的前言部分已经说过，本书的目的主要有两个：一是详细深入地讲解C#的语法和语义，二是阐述各种.NET API的用法，包括利用ADO.NET访问数据库、Entity Framework、LINQ技术、WPF、WCF、WF，以及运用ASP.NET进行Web站点开发。常言道："千里之行，始于足下。"欢迎从本章开始你的"千里之行"。

本章将对本书其余部分所涉及的各个方面做概念性的描述。最开始将从宏观上讨论一些.NET相关主题，如程序集、CIL（Common Intermediate Language，公共中间语言）和JIT（just-in-time，即时）编译。接下来，除了预览C#语言的一些主要功能之外，还将讲述.NET Framework不同方面，例如CLR（公共语言运行库）、CTS（公共类型系统）和CLS（公共语言规范）之间的关系。

本章还会探讨.NET 4.5基础类库提供的功能，基础类库的英文Base Class Library缩写为BCL。本章最后概述.NET平台的语言无关性和平台无关性（别惊讶，.NET并不局限于Windows操作系统），还会大体讲一下在Windows 8操作系统下构建应用程序时.NET的角色。当然，所有这些主题都将在本书其余部分详细探讨。

1.1 初识.NET 平台

在微软发布C#语言和.NET平台之前，为Windows操作系统家族创建应用程序的开发者常常使用COM编程模型。COM（Component Object Model，组件对象模型）允许个人构建可由不同编程语言共享的代码库。例如，Visual Basic开发者可以使用C++程序员构建的COM库。COM的语言无关特点自然十分有用，但它复杂的基础结构、脆弱的部署模型常常带来很多麻烦，并且只能部署在Windwos操作系统上。

尽管COM有很多复杂性和局限性，但不计其数的应用程序还是成功地构建于这个基础结构之上。然而在今天，大多数为Windows操作系统家族创建的应用程序都不是用COM模型构建的。桌面应用、网站、操作系统服务、数据访问或业务逻辑复用库都是使用.NET平台创建的。

.NET平台的主要优点

前面提到过，C#和.NET平台是2002年正式发布的，当时主要为了提供一种比COM更强大、更灵活、更简洁的编程模型。从本书后面的内容你会看到，.NET Framework用于在Windows系列操作系统和其他如Mac OS X或Unix/Linux等非微软的操作系统中创建系统。为了打好基础，我们先来快速浏览一下.NET Framework的一些核心功能。

- **对已有代码具有完全的互操作性**：这（当然）是一件很好的事情。已有的COM二进制组件可以和更新的.NET二进制组件共存，反之亦然。在.NET 4.0及后续版本，使用dynamic关键字（详见第16章）可以进一步简化这种互操作性。
- **支持多种编程语言**：使用多种编程语言（C#、Visual Basic、F#等）创建.NET应用。
- **所有支持.NET的语言共享的公共运行时引擎**：这个引擎的一个特点是具有一组明确定义的类型，而每一种支持.NET的语言都能“明白”这些类型。
- **语言集成**：.NET支持跨语言的继承、异常处理和代码调试。比方说，C#中定义的基类可以在Visual Basic进行扩展。
- **全面的基础类库**：这个库除隐藏了原始API调用的复杂性外，还提供了被所有支持.NET的语言所使用的一致的对象模型。
- **简化的部署模型**：与COM不同，.NET库不需要将二进制单元注册到系统注册表了。另外，.NET允许同一个*.dll的不同版本存在于同一台机器上。

这些核心功能以及更多内容会在后面的章节详细介绍。

1.2 .NET 平台构造块（CLR、CTS 和 CLS）简介

了解了.NET的优点之后，让我们来预览一下使.NET成为现实的3个关键（而且相互关联的）实体：CLR、CTS和CLS。从程序员的角度看，.NET可以理解为一个运行库环境和一个全面的基础类库。运行库层的正式名称是CLR（Common Language Runtime，公共语言运行库）。其主要作用是为我们定位、加载和管理.NET类型，同时也负责一些低层细节的工作，如内存管理、应用托管、处理线程、安全检查等。

.NET平台的另一个构造块是CTS（Common Type System，公共类型系统）。CTS规范完整描述了运行库所支持的所有可能的数据类型和编程结构，指定了这些实体间如何交互，也规定了它们在.NET元数据格式中的表示（本章后面将会给出更多关于元数据的信息，第15章将详细介绍这方面的内容）。

要注意的是，一种特定的支持.NET的语言可能不支持CTS所定义的所有特性。CLS（Common Language Specification，公共语言规范）是一个相关的规范，定义了一个让所有.NET语言都支持的公共类型和编程结构的子集。这样，如果构造的.NET类型仅公开与CLS兼容的特性，那么可以肯定其他所有支持.NET的语言都能使用它们。反之，如果使用了与CLS不兼容的数据类型或编程结构，就不能保证所有的.NET语言能和你的.NET代码库交互。庆幸的是，如你在本章后面所看到的那样，让C#编译器遵从CLS来验证代码是十分简单的。

1.2.1 基础类库的作用

除了CLR和CTS/CLS规范之外，.NET平台提供了一个适用于全部.NET程序语言的基础类库（BCL）。这个基础类库不仅封装了各种基本类型，如线程、文件输入/输出（I/O）、图形绘制以及与各种外部硬件设备的交互，还支持在实际应用中用到的一些服务。

例如，基础类库定义了一些可创建任意类型软件应用程序的类型，例如，使用ASP.NET创建Web站点，使用WCF创建网络服务，使用WPF创建桌面GUI应用程序，等等。基础类库还定义了另外一些类型，可以与特定计算机上的XML文档、本地目录和文件系统互动，通过ADO.NET与关系数据库交流，等等。如图1-1所示，可以从宏观上看到CLR、CTS、CLS和基础类库之间的关系。

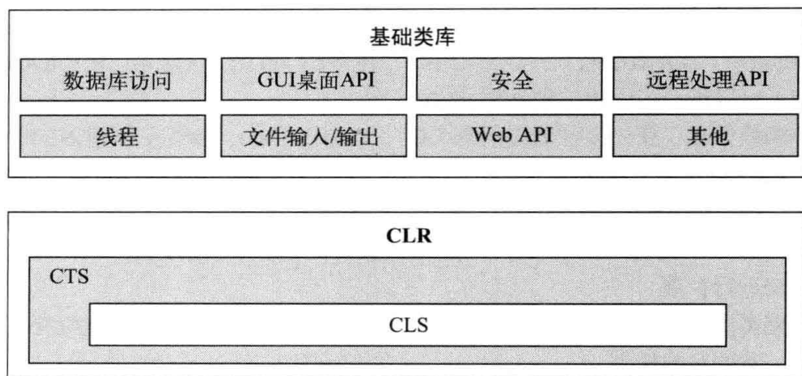


图1-1　CLR、CTS、CLS和基础类库之间的关系

1.2.2　C#的优点

C#的核心语法和Java语法很相似。然而，并不能说C#抄袭了Java。本质上，C#和Java都属于C语言系列（包括C、Objective C、C++等），它们有类似的语法。

实际上，C#的许多语法结构与VB和C++的很多方面都有渊源。例如，与VB类似，C#支持类型属性（property）[①]（与传统的获取方法和设置方法相反）和可选参数。与C++类似，C#允许重载操作符，且支持创建结构、枚举和回调函数（使用委托）。

此外，在浏览本书时，我们很快就会看到C#支持各种函数式语言（如LISP或Haskell）中的很多特性，例如Lambda表达式和匿名类型。此外，由于LINQ的出现，C#支持很多编程结构，在编程语言中显得非常独特。尽管如此，C#的核心始终受到C系列语言的影响。

C#是多种语言的混合体，因此它像Java一样语法简洁，像VB一样使用简单，像C++一样功能强大和灵活。以下是C#核心特征的一部分，其中大部分特点也是其他支持.NET的程序语言所共有的特征。

- 不需要指针！C#程序通常不需要直接对指针进行操作（尽管在绝对必要时也能自由地进行底层操作，详见第11章）。
- 垃圾收集器能够自动管理内存。因此，C#不支持`delete`关键字。
- 类、接口、结构、枚举和委托都有正式的语法结构。
- 具有与C++类似的功能，可以简单地重载操作符为自定义类型（例如，不需要操心确保“返回`*this`以能够链接”）。
- 支持基于特性的编程。这种方式的开发允许我们注释类型及其成员来进一步限定其行为。例如，用这个`[Obsolete]`特性标记某种方法后，后面再使用这种方法的时候就会打印自定义的警告信息。

① 在C#中，property（属性）是一种特殊的字段（内部用`get`/`set`方法实现，外部调用则与普通字段无异），而attribute（特性，也常译为属性或性质）是一种为各种语言构造添加元数据信息的方式，二者与一般场合下所说的属性（往往指普通类字段）和特性（指一般意义上的功能）都不同（本书已尽量避免使用后两种说法），请读者注意区分。
——编者注

随着.NET 2.0的发布(大约在2005年),C#编程语言被更新以支持很多花哨的东西,主要是以下几项。

- 构建泛型类型和泛型成员的能力。使用泛型,我们可以构建非常高效的并且类型安全的代码,在和泛型项交互的时候可以定义很多占位符。
- 支持匿名方法,它允许我们在任何需要委托类型的地方提供内联函数。
- 使用partial关键字跨多个代码文件定义单个类型的能力(或者如果有必要的话,可以作为内存中的表示)。

.NET 3.5(大约发布于2008年)为C#编程语言增加了更多功能,包括如下特性。

- 支持强类型的查询(如LINQ),可用于和各种形式的数据进行交互。从第12章开始讲解LINQ。
- 支持匿名类型,它允许我们建模一个类型的形(shape)而不是其行为。
- 使用扩展方法扩展既有类型(没有子类)功能的能力。
- 包含了Lambda操作符(=>),它可以进一步简化.NET委托类型的使用。
- 新的对象初始化语法,它允许我们在创建对象时设置属性的值。

.NET 4.0(2010年发布)再次为C#添加了少量特性,下面举几个例子。

- 支持可选的方法参数和命名的方法参数。
- 支持通过dynamic关键字在运行时动态查找成员。第18章提供了一个统一的方法用于在运行时调用成员,而不必理会成员的实现框架(COM、IronRuby、IronPython或通过.NET反射服务)。
- 泛型类型的操作将更加直观,因为你可以使用协变和逆变,轻易地在泛型数据和普通的System.Object集合之间进行相互映射。

最后是随.NET 4.5发布的C#当前版本。C#当前版本提供了一对关键字(async和await),极大地简化了多线程和异步编程。如果你使用过以前版本的C#,一定会记得通过副线程调用方法需要大量的含义模糊的代码,并使用不同的.NET命名空间。而现在C#提供了语言关键字来为我们处理这种复杂性,异步调用方法的过程几乎像以同步方式调用方法一样简单。第19章将详细介绍这些话题。

1.2.3 托管代码与非托管代码

关于C#语言,要理解的最重要的一点可能是,它生成的代码只能在.NET运行库中执行(你不能用C#来构建本机的COM服务器或非托管的C/C++ API应用程序)。正式的说法是,这种必须在.NET运行库下执行的代码称为托管代码(managed code)。这些包含托管代码的二进制单元称为程序集(assembly),稍后将详细介绍。反之,不能直接在.NET运行库承载(host)的代码称为非托管代码(unmanaged code)。

1.3 其他支持.NET 的编程语言

应该知道,C#并不是构建.NET应用的唯一一种语言。在你安装Visual Studio时,你将得到5种托管语言:C#、Visual Basic、C++/CLI、JavaScript和F#。

说明 F#是一门基于函数式语言的.NET语言。尽管F#可以看成是一门纯粹的函数式语言,但它同样也支持OOP和.NET基础类库。如果你想深入了解这门托管语言,可以访问F#官方网页http://msdn.microsoft.com/fsharp。

除了微软提供的托管语言之外，还有Smalltalk、Ruby、Python、COBOL和Pascal的.NET编译器等。尽管本书以C#为主，但你可能有兴趣关注下面的网站：

http://www.dotnetlanguages.net

如果单击这个网站主页顶部的Resources链接，你会看到大量.NET编程语言的列表和相关链接，你可以从这里下载各种编译器（如图1-2所示）。

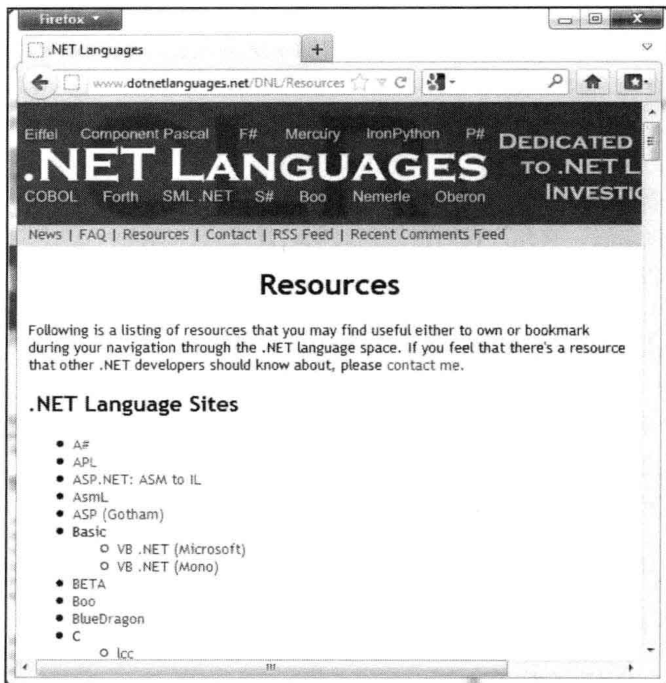


图1-2　DotNetLanguages.net中包含已知.NET编程语言相关的资源，类似网站还有很多

如果你对使用C#语法构建.NET 程序感兴趣，我还是建议你访问这个网站，因为你肯定会发现许多.NET语言值得去研究（如LISP.NET等）。

多语言世界中的生活

当初步了解.NET的语言无关性后，开发者会提出许多问题。其中最普遍的问题可能就是："如果所有的.NET语言都会编译成'托管代码'，为什么我们还需要多种编译器呢？"

这个问题有多个答案。首先，程序员在选择编程语言时有各自不同的喜好（包括我自己）。一些人喜欢充满分号和圆括号而关键字相当少的语言，另一些人喜欢更具有可读性语法标记的语言（如VB），还有一些人在开始转向.NET平台时还希望可以使用他们已掌握的技能（通过COBOL .NET编译器）。

现在，平心而论，如果微软推出一门派生自BASIC语言系列的"官方".NET语言，你认为所有的程序员会喜欢这样的选择吗？或者，如果这个唯一的"官方".NET语言是基于Fortran语法的，那么可以想象所有人都会对.NET置之不理。因为.NET运行库并不在意一段托管代码是由哪种语言生成的，

所以.NET程序员可以继续使用他们熟悉的语法，且与组员、部门甚至其他公司共享编译的程序集（不管他们用的是哪种.NET语言）。

将各种.NET语言集成为一个统一软件方案的另一个好处，就是能够取长补短。所有的编程语言都有各自的优点和缺点。例如，一些编程语言对高级的数学处理有相当完美的内在支持能力。另一些则精于支持财务计算、逻辑计算和与大型机交互等。当你学习到某种编程语言的优点并将其融合到.NET平台时，大家就都能受益。

当然，实际上我们大部分时间还是在用自己习惯的.NET语言来编写程序。但是，一旦学会了一种.NET语言的语法，就很容易掌握其他的了。这是非常有益的，对软件技术顾问而言尤其如此。如果你熟悉C#，在为只使用Visual Basic的客户做咨询时，你仍然能够使用.NET Framework的功能，并且可以毫不费力地掌握代码的整体结构。够棒的吧。

1.4 .NET 程序集概览

不管选择了哪种.NET语言编程，需要明白的是，尽管.NET二进制文件与非托管Windows二进制文件（*.dll或*.exe）具有相同的文件扩展名，但它们的内部却是完全不同的。具体说来，.NET二进制文件不包含特定于平台的指令，它包含的是平台无关的IL（Intermediate Language，中间语言）和类型元数据。图1-3清楚显示了这个流程。

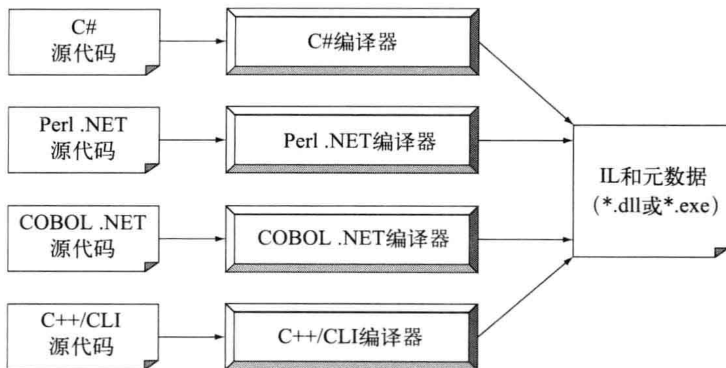


图1-3 所有支持.NET的编译器都生成IL指令和元数据

说明 需要说明的一点是IL的缩写。在.NET的开发过程中，IL的官方术语是MSIL（微软中间语言）或CIL（公共中间语言）。因此，当阅读.NET资料时，你应该明白IL、MSIL和CIL指的是同一个概念。本书谈到底层指令集时使用缩写CIL。

当使用支持.NET的编译器生成*.dll或*.exe文件时，二进制大对象会被打包成一个程序集。本书将在第14章详细介绍.NET程序集。然而为了方便讨论.NET运行库环境，你需要理解这个新文件格式的基本属性。

前文提到，程序集包含CIL代码，后者在概念上类似于Java的字节码，因为它只在绝对必需的情况下才编译为特定平台的指令。“绝对必需”通常是指一段CIL指令（例如一个方法实现）被.NET 运行库引用时。

除了CIL指令外，程序集还包含元数据（metadata）。元数据详尽描述了二进制文件中每个“类型”的特征。例如，一个名为SportsCar的类，这个类型的元数据描述一些详细信息，比如SportsCar的基类，这个基类如果有接口，则其接口由SportsCar来实现，元数据同时也详细描述了由SportsCar类型支持的各种成员。.NET元数据总是存在并且会由某种支持.NET的编译器自动生成。

最后，除了CIL和类型元数据之外，程序集本身也使用元数据进行描述，这类元数据的正式名称是清单（manifest）。清单记录了程序集的当前版本信息、文化信息（用于本地化字符串和图像资源）和正确执行所需的外部引用程序集的列表。在接下来的几章中，将使用各种各样的工具来检验程序集的类型、元数据和清单信息。

1.4.1　CIL的作用

现在我们来深入探讨CIL代码、类型元数据和程序集清单。CIL是一种和平台无关的语言。例如，下列的C#代码构成一个简单的计算器。现在不必在意具体的语法，只要注意Calc类中的Add()方法的格式：

```
// Calc.cs
using System;

namespace CalculatorExample
{
  // 这个类包含应用程序的入口点
  class Program
  {
    static void Main()
    {
      Calc c = new Calc();
      int ans = c.Add(10, 84);
      Console.WriteLine("10 + 84 is {0}.", ans);

      // 等待用户按Enter键来结束程序
      Console.ReadLine();
    }
  }

  // C#计算器
  class Calc
  {
    public int Add(int x, int y)
    { return x + y; }
  }
}
```

C#编译器（csc.exe）编译这段代码后，就会得到一个单文件*.exe程序集，其中包含一个程序集清单、CIL指令和描述Calc与Program类的各方面信息的元数据。

说明　第2章讲述了使用C#编译器编译代码的细节和图形IDE的使用，如Microsoft Visual Studio。

例如，如果用ildasm.exe（本章稍后会说明）打开该程序集，会发现Add()方法被CIL表示为：

```
.method public hidebysig instance int32 Add(int32 x,
  int32 y) cil managed
{
  //代码长度 9(0x9)①
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldarg.2
  IL_0003:  add
  IL_0004:  stloc.0
  IL_0005:  br.s IL_0007
  IL_0007:  ldloc.0
  IL_0008:  ret
} // Calc::Add方法结束
```

如果看不懂这段CIL代码，也不必担心，第17章会讲述CIL编程语言的基础。需要注意一点，C#编译器生成的是CIL，并不是平台相关的指令。

这一点适用于所有支持.NET的编译器。为了便于说明，我们假设用VB创建一个和上面C#相同的程序：

```
' Calc.vb
Imports System

Namespace CalculatorExample

  ' VB“模块”是一个只包含静态成员的类
  Module Program
    Sub Main()
      Dim c As New Calc
      Dim ans As Integer = c.Add(10, 84)
      Console.WriteLine("10 + 84 is {0}.", ans)
      Console.ReadLine()
    End Sub
  End Module

  Class Calc
    Public Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
      Return x + y
    End Function
  End Class

End Namespace
```

如果查看Add()方法的CIL指令，你会觉得它与VB编译器vbc.exe生成的代码非常相似（只有很少的差别）：

```
.method public instance int32 Add(int32 x,
  int32 y) cil managed
{
  //代码长度 8(0x8)
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:  ldarg.1
  IL_0001:  ldarg.2
  IL_0002:  add.ovf
```

① 编译生成的原始CIL代码中注释本身为英文，书中翻译是为了阅读方便，请读者留意。——编者注

```
    IL_0003:  stloc.0
    IL_0004:  br.s IL_0006
    IL_0006:  ldloc.0
    IL_0007:  ret
  } // Calc::Add方法结束
```

源代码　Calc.cs和Calc.vb代码文件位于Chapter 1子目录下。

1. CIL的好处

到此，你可能很想弄清楚，不直接把源代码编译为特定的指令集而是编译为CIL的好处到底在哪里。有一点好处就是语言的集成性。如前所述，每种支持.NET的编译器生成的是几乎完全相同的CIL指令。因此，所有语言都能很好地在定义明确的二进制文件间交互。

此外，CIL是平台无关的，.NET Framework本身也是平台无关的。Java程序员早已体会到了这一点好处（例如，一个代码库就可以在多种操作系统上运行）。实际上，已经存在C#语言的国际标准和大量的.NET平台和实现的子集，它们可以供许多非Windows的操作系统使用（本章最后会详细说明）。

2. 将CIL编译成特定平台的指令

由于程序集包含的是CIL指令而不是某一特定平台的指令，CIL代码必须在使用之前进行即时编译。将CIL代码编译成有意义的CPU指令的工具称为JIT（即时）编译器，有时也称为Jitter。.NET运行库环境将使用针对各种不同CPU的JIT编译器，每个编译器都会针对底层平台进行优化。

比如，在手持设备（如Windows移动设备）上部署一个.NET应用程序，就可以配备相应的Jitter以在低内存环境下运行。另一方面，如果为后台服务器部署程序集（通常内存不是问题），那么Jitter又能进行优化，使代码在高内存环境下运行。这样，开发人员只需编写一套代码，就能在不同体系结构的设备上通过JIT编译器高效地编译和执行。

另外，当给定的Jitter编译器将CIL指令编译为相应的机器代码时，它会用适合目标操作系统的方式将结果缓存在内存中。这样，如果`PrintDocument()`方法被调用，则它对应的CIL指令将在第一次调用中被编译成特定平台的指令并被保留在内存中以备以后使用。因此，在下一次调用`PrintDocument()`时，就不需要编译CIL了。

说明　同样可以使用.NET 4.5 Framework SDK附带的ngen.exe命令行工具在安装程序时执行“预JIT”。这样做可以改善图形密集的应用程序的启动时间。

1.4.2　.NET类型元数据的作用

除了CIL指令以外，.NET程序集还包括全部完整且准确的元数据，这些元数据描述了每一个二进制文件中定义的类型（如类、结构、枚举等）以及每个类型的成员（比如属性、方法和事件等）。值得庆幸的是，生成最新的和最大的类型元数据总是编译器的工作而不是程序员的工作。因为.NET元数据非常详细，所以程序集完全成了自描述的实体。

为了说明.NET类型元数据的格式，我们来看之前为C# `Calc`类的`Add()`方法生成的元数据（为VB版本的`Add()`方法生成的元数据是相似的）：

```
TypeDef #2 (02000003)
-------------------------------------------------------
  TypDefName: CalculatorExample.Calc  (02000003)
  Flags     : [NotPublic] [AutoLayout] [Class]
  [AnsiClass] [BeforeFieldInit]  (00100001)
  Extends   : 01000001 [TypeRef] System.Object
  Method #1 (06000003)
-------------------------------------------------------
  MethodName: Add (06000003)
  Flags     : [Public] [HideBySig] [ReuseSlot]  (00000086)
  RVA       : 0x00002090
  ImplFlags : [IL] [Managed]  (00000000)
  CallCnvntn: [DEFAULT]
  hasThis
  ReturnType: I4
    2 Arguments
    Argument #1:  I4
    Argument #2:  I4
    2 Parameters
    (1) ParamToken : (08000001) Name : x flags: [none] (00000000)
    (2) ParamToken : (08000002) Name : y flags: [none] (00000000)
```

元数据不仅用于.NET运行库环境的许多方面，而且用于各种开发工具中。例如，诸如Visual Studio等工具提供的智能感知（IntelliSense）特性就能在设计阶段读取程序集的元数据。各种对象浏览工具、调试工具以及C#编译器自身都使用元数据。需要注意的是，元数据是许多.NET技术的支柱，这些技术包括WCF、反射、晚期绑定和对象序列化。第15章将正式讨论.NET元数据的作用。

1.4.3 程序集清单的作用

最后，请记住.NET程序集也包含描述程序集自身的元数据（称为清单，manifest）。在许多细节中，清单记录了所有确保现有程序集正常工作的外部程序集、程序集的版本号、版权信息等。同类型元数据一样，生成程序集清单也是编译器的工作。下面是编译Calc.cs代码文件（本章前面提到过）时所生成的清单的一些重要细节（假设我们指示编译器将程序集命名为Calc.exe）：

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
.assembly Calc
{
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module Calc.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 0x00000200
.corflags 0x00000001
```

简要地说，这个清单记录了Calc.exe（通过`.assembly extern`指令）所需要的外部程序集，同时也记录了程序集本身的各种特性（如版本号、模块名称等）。第14章将详细介绍清单数据的优点。

1.5 CTS

一个给定的程序集可能包含任意数量的不同“类型”。在.NET领域，类型（type）是一个一般性的术语，它指的是集合{类，接口，结构，枚举，委托}里的任意一个成员。当用支持.NET的语言构建解决方案时，很有可能要与这些类型打交道。例如，程序集可能定义了一个类，它又实现了一些接口。或许其中某个接口方法采用枚举类型作为输入参数，而在调用时返回一个结构。

CTS（公共类型系统）是一个正式的规范，它规定了类型必须如何定义才能被CLR承载。通常，只有那些创建针对.NET平台的工具或编译器的人才对CTS的内部工作非常关心。但是，对于所有.NET编程人员来说，学习如何在自己使用的语言中使用由CTS定义的5种类型，是非常重要的。这里简单概括一下。

1.5.1 CTS类类型

每一种支持.NET的语言至少要支持类类型（class type）的概念，这是OOP的基础。类可能由很多成员（诸如构造函数、属性、方法和事件）和数据点（字段）组成。在C#中，使用class关键字来声明类：

```
// 带有1个方法的C#类类型
class Calc
{
  public int……
  {
    return……
  }
}
```

第5章将讲述用C#构造CTS类类型的过程。表1-1给出了有关类类型的一些特征。

表1-1　CTS类类型

类的特征	在生命周期里的意义
类是否被“密封”	密封类不能作为其他类的基类
类实现任何接口了吗	接口是抽象成员的集合，它在对象和对象的用户间提供一个契约。CTS允许类实现任何数目的接口
类是具体的还是抽象的	抽象类是不能直接创建的，但是可以用来为派生类型定义公共的行为。具体类可以直接创建
这个类的可见性是什么	每个类必须用关键字（比如public或internal）设置可见性。基本上，可见性定义了该类是被外部程序集使用，还是仅能在定义了它的程序集中使用

1.5.2 CTS接口类型

接口（interface）就是由抽象成员定义所组成的一个具名集合，可通过一个给定的类或结构来实现。在C#中，接口类型使用interface关键字来定义。一般情况下，所有的.NET接口均以大写字母I开头，例如：

```
// C#接口通常被声明为公共的，这样其他程序集中的类型就可以实现其行为
public interface IDraw
{
  void Draw();
}
```

就它们自身而言，接口没有什么用。然而，当一个类或结构用其独特方式来实现一个给定接口时，

你将能够以多态方式通过接口引用来请求使用所提供的功能。基于接口的编程将在第8章中全面介绍。

1.5.3 CTS结构类型

CTS中还支持结构（structure）的概念。如果你有C语言的背景，应该会很高兴地发现这种用户自定义类型（UDT）也存在于.NET领域中（虽然它们的行为在底层不同）。简单地说，结构（struct）可以看做是具有值语义的轻量级类类型。关于结构的细节，请参见第4章。通常，结构最适合建模几何和数学数据。在C#中，通常使用struct关键字创建结构：

```
// C#结构类型
struct Point
{
  // 结构可以包含字段
  public int xPos, yPos;

  // 结构可以包含参数化构造函数
  public Point(int x, int y)
  { xPos = x; yPos = y;}

  // 结构可以定义方法
  public void PrintPosition()
  {
    Console.WriteLine("({0}, {1})", xPos, yPos);
  }
}
```

1.5.4 CTS枚举类型

枚举（enumeration）是一种便利的编程结构，它可以用来组成名称/值对。例如，假设你在开发一个视频游戏的程序，要让玩家在3种角色（Wizard、Fighter或Thief）中选择一个。你完全可以用enum关键字来建立一个自定义的枚举，而不用老是要记着代表每种可能性的原始数字值：

```
// C#枚举类型
enum CharacterType
{
  Wizard = 100,
  Fighter = 200,
  Thief = 300
}
```

在默认情况下，每一项是用一个32位的整数来存储的，但如果需要，也可以改变存储大小（例如，在为Windows移动设备之类的低内存设备编程时）。另外，CTS要求枚举类型派生自基类System.Enum。在第4章中你将看到，这个基类定义了一些有趣的成员，允许通过编程提取、操作和变换底层的名称/值对。

1.5.5 CTS委托类型

委托（delegate）在.NET中等效于类型安全的C风格的函数指针。它们的主要不同在于，.NET委托是派生自System.MulticastDelegate的类，而不是一个简单地指向原始内存地址的指针。在C#中，委托是使用关键字delegate来声明的：

```
// 这个C#委托类型可以“指向”任意带有两个整型参数且返回一个整型值的方法
delegate int BinaryOp(int x, int y);
```

一个实体可以用委托向另一个实体传递调用，另外，委托也为.NET事件架构提供了基础。在第11章和第19章中，你将看到，委托对多路广播（即将一个请求转发给多个接收者）和异步方法调用（即从另一个线程调用方法）有着内在支持。

1.5.6　CTS类型成员

现在你已经看到了由CTS正式规定的各种类型，但还要认识到，大部分的类型可以含有任意数量的成员。说得更正式一些，类型成员是集合{构造函数，终结器，静态构造函数，嵌套类型，操作符，方法，属性，索引器，字段，只读字段，常量，事件}中的元素之一。

CTS定义了各种可能与具体成员关联的修饰语（adornment）。例如，每个成员都有一个给定的可见性特征（如公共的、私有的和受保护的等）。有些成员可能被声明成抽象的以加强派生类的多态性，有些成员可声明为虚拟的以定义一个封装（但可重写）的实现。同样，绝大部分成员可设置成静态的（在类级别绑定）或者实例（在对象级别绑定）。类型成员的创建会在以后的几章中介绍。

说明　在第9章中将讲到，C#语言也支持泛型类型和泛型成员的创建。

1.5.7　内建的CTS数据类型

CTS需要关注的最后一个方面是，它建立的一套定义明确的核心数据类型。尽管不同的语言通常都有自己唯一的用于声明内建CTS数据类型的关键字，但是所有语言的关键字最终将解析成定义在mscorlib.dll程序集中的相同类型。参考表1-2，它描述了如何在不同的.NET语言中表示关键的CTS数据类型。

表1-2　内建的CTS数据类型

CTS数据类型	VB关键字	C#关键字	C++/CLI关键字
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int 或 long
System.Int64	Long	long	__int64
System.UInt16	UShort	ushort	unsigned short
System.UInt32	UInteger	uint	unsigned int或unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	Float
System.Double	Double	double	double
System.Object	Object	object	object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	bool

由于各种托管语言的关键字只是System命名空间中真实类型的简化符号，我们不需要担心数值数据的上溢或下溢，或是字符串和布尔型数据在内部是怎样跨不同语言进行表示的。下面的代码片段使用C#和VB，通过语言关键字和正式的CTS数据类型分别定义了32位数值变量。

```
// 用C#定义整型数据
int i = 0;
System.Int32 j = 0;

' 用VB定义整型数据
Dim i As Integer = 0
Dim j As System.Int32 = 0
```

1.6 CLS

我们知道，不同的语言往往用不同的、语言特定的术语表达相同的程序构造，比如，在C#中使用加号（+）操作符表示字符串拼接，而在VB中却使用“&”符号。即使两种不同的语言表达相同的编程惯用法（比如一个不返回值的函数），在表面看起来，语法也可能非常不同。

```
// C#不返回值的方法
public void MyMethod()
{
  // 一些有趣的代码
}

' VB不返回值的方法
Public Sub MyMethod()
  ' 一些有趣的代码
End Sub
```

正如你已经看到的，在.NET运行库看来这些较小的语法变化是微不足道的，因而不同的编译器（这里用到的是vbc.exe或csc.exe）将产生类似的CIL指令集。然而，语言也可能在功能上不同，比如，.NET语言可能有也可能没有关键字来表示无符号数据，可能支持也可能不支持指针类型。对于这些可能的变化，理想情况是所有支持.NET的语言都有一个可以遵循的基准。

CLS就是这样一套规则，它清晰地描述了支持.NET的编译器必须支持的最小的和完全的特征集，以生成可由CLR承载的代码，同时可以被基于.NET平台的其他语言用统一的方式进行访问。CLS可以看成是由CTS定义的完整功能的一个子集。

如果打算让自己的产品功能无缝地融合到.NET世界，那么CLS是编译器创建者最终必须遵循的一套规则。每个规则被赋予一个简单的名字（如CLS规则6），描述了这个规则如何影响创建编译器的人以及（以某种方式）与他们交互的人。影响最大的是规则1。

- 规则1：CLS规则仅适用于类型中向定义它的程序集以外公开的部分。

根据这个规则，可以（正确地）推断其余的CLS规则对于用来建立一个.NET类型内部运行功能的逻辑是不适用的。必须遵循CLS的类型的唯一一点，就是成员定义本身（即命名规范、参数和返回类型）。成员的实现逻辑可以使用其他的非CLS技术，程序外部并不知道这些不同。

举例说明，下面的`Add()`方法就没有遵循CLS规则，因为它的参数和返回值使用了无符号数（无符号数不符合CLS）：

```
class Calc
{
  // 公开的无符号类型数据不遵循CLS规则
  public ulong Add(ulong x, ulong y)
  {
    return x + y;
  }
}
```

然而，如果像下面一样在程序内部使用无符号数：

```
class Calc
{
  public int Add(int x, int y)
  {
  // 当ulong类型变量仅仅在内部使用时，仍然遵循CLS规则
    ulong temp = 0;
    ...
    return x + y;
  }
}
```

这仍然遵循CLS规则，可以保证所有的.NET语言都能调用Add()方法。

当然，除规则1外，CLS还定义了很多其他的规则。例如，CLS描述了一种语言如何表示文本字符串，如何在内部表示枚举（用于存储的基类型），如何定义静态成员，等等。好在你不需要记忆所有的规则也能成为精通.NET的程序员。总地来说，只有那些工具/编译器的开发人员才会对CTS和CLS规范的具体细节感兴趣。

确保遵循CLS

正如本书将提到的，C#定义了一些不遵循CLS规则的程序结构，但你仍然可以使用一个专门的.NET特性指示C#编译器检查代码是否遵循CLS规则。

```
// 指示C#编译器检查是否遵循CLS规则
[assembly: CLSCompliant(true)]
```

第15章会详细探讨基于特性的编程。目前，只需要知道[CLSCompliant]特性就是用来指示C#编译器按CLS规则检查每行代码的。如果代码违反了CLS，就会给出编译器错误和关于错误代码的描述。

1.7　CLR

除CTS和CLS规范外，我们现在要了解的最后一个字母缩写术语是CLR。从编程角度来说，运行库（runtime）可以理解为执行给定编译代码单元所需的外部服务的集合。比如，当Java程序员向一台新电脑部署软件时，要确保软件运行，电脑上就要安装JVM（Java Virtual Machine，Java虚拟机）。

.NET平台提供了另一种运行库系统。.NET运行库与刚才提到的其他运行库的关键不同在于，.NET运行库提供了一个定义明确的运行库层，可以被支持.NET的所有语言和平台所共享。

CLR中最重要的部分是由名为mscoree.dll的库（又称公共对象运行库执行引擎）物理表示的。当用户程序引用一个程序集，要使用它时，mscoree.dll将首先自动加载，然后由它负责将需要的程序集导入内存。运行时引擎负责许多任务，首要的任务是负责解析程序集的位置，并通过读取其中包含的元数据，在二进制文件中发现所请求的类型。接着，CLR在内存中为类型布局，将关联的CIL编译成特定平台的指令，执行所有需要的安全检查，然后运行当前的代码。

除了导入自定义的程序集和建立自定义的类型，必要时CLR也会与包含在.NET基础类库的类型交互。虽然完整的基础类库被分为若干分离的程序集，但最重要的程序集是mscorlib.dll。mscorlib.dll包含大量核心类型，它们封装了各种常见的编程任务与.NET语言用到的核心数据类型。当建立一个.NET解决方案时，你可以自动访问这些程序集。

图1-4说明了发生在源代码（它使用了许多基础类库类型）、.NET编译器和.NET执行引擎之间的工作流。

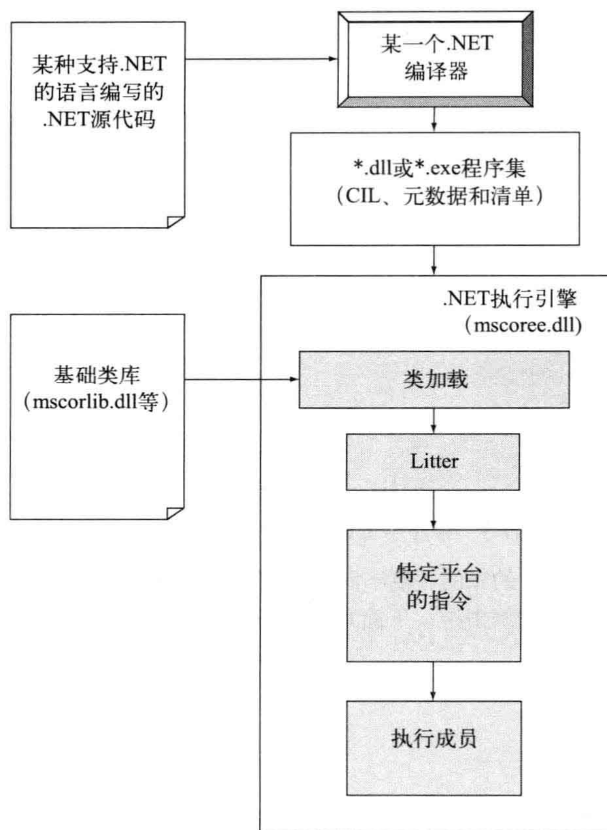


图1-4 mscoree.dll工作流

1.8 程序集/命名空间/类型的区别

我们都知道代码库的重要性，框架库的关键就是要给程序员提供一套定义明确的既有代码，从而显著提高开发效率。C#没有提供特定语言的代码库，但C#程序员可以利用语言无关的.NET代码库。为确保基础类库中的所有类型能良好地组织在一起，.NET平台提出了命名空间（namespace）的概念。

简单地讲，命名空间就是一个程序集内相关类型的一个分组。举例来讲，`System.IO`命名空间包含了有关文件I/O的类型，`System.Data`命名空间定义了基本的数据库类型，等等。需要特别指出的是，一个程序集（比如mscorlib.dll）可以包含任意个命名空间，每个命名空间又可以包含多种类型。

为了更清楚地阐述，图1-5展现了一个Visual Studio Object Browser的截图。这个工具可以用来检查当前项目引用的程序集、位于一个特定程序集中的命名空间、给定命名空间中的类型以及具体类型的成员。注意，mscorlib.dll包含了许多不同的命名空间（如`System.IO`），每个命名空间都拥有语义上相关的类型（如`BinaryReader`）。

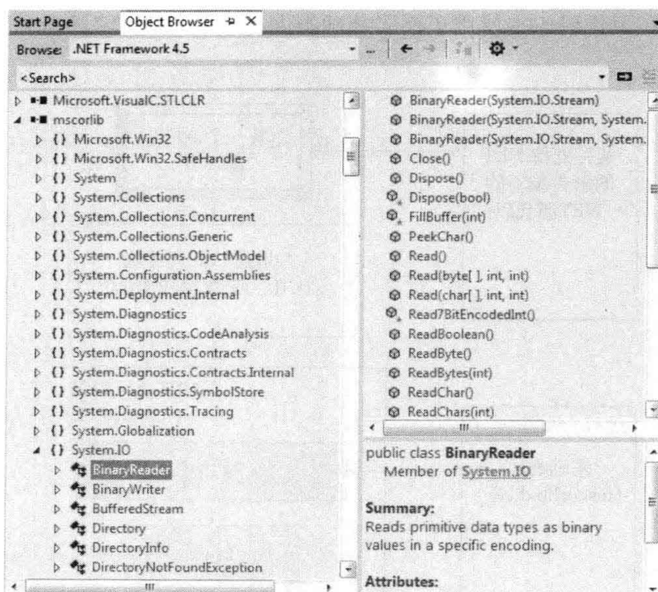


图1-5 程序集可以包含任意多个命名空间

这种方法和一个特定于语言的库的关键不同在于，任何基于.NET运行库的语言都可以使用相同的命名空间和相同的数据类型。举例来讲，下面3个程序分别使用了C#、VB和C++/CLI编写，演示了常见的“Hello World”应用程序。

```
// 用C#写的Hello World
using System;

public class MyApp
{
  static void Main()
  {
    Console.WriteLine("Hi from C#");
  }
}
```

```
' 用VB写的Hello World
Imports System
Public Module MyApp
  Sub Main()
    Console.WriteLine("Hi from VB")
  End Sub
End Module
```

```
// 用C++/CLI写的Hello World
#include "stdafx.h"
using namespace System;

int main(array<System::String ^> ^args)
{
  Console::WriteLine(L"Hi from C++/CLI");
  return 0;
}
```

注意，每种语言都使用了System命名空间中定义的Console类。除了语法上略微不同外，3个应用程序从外观上和逻辑上看起来非常相似。

应该清楚，.NET程序员的主要目标就是逐步了解大量定义在.NET命名空间里的类型。最基本的命名空间无疑是System。这个命名空间提供了大量核心的类型，是.NET程序员会反复使用的。实际上，因为核心数据类型（System.Int32、System.String等）是在System命名空间中定义的，所以如果完全不引用System命名空间，就根本无法开发C#应用程序。表1-3简要介绍了一些（当然不是全部的）按相关功能分组的.NET命名空间。

表1-3 .NET命名空间举例

.NET命名空间	作　用
System	在System内，你将会发现很多有用的类型，可以用来处理内建数据、数学计算、随机数的产生、环境变量、垃圾收集器以及一些常见的异常和特性
System.Collections System.Collections.Generic	这些命名空间定义了一些集合容器类型，还有一些基类型和接口，使你有可能构建自定义的收集器
System.Data System.Data.Common System.Data.EntityClient System.Data.SqlClient	这些命名空间用来使用ADO.NET与数据库交互
System.IO System.IO.Compression System.IO.Ports	这些命名空间定义了许多处理文件I/O、数据压缩和端口操作的类型
System.Reflection System.Reflection.Emit	这些命名空间定义了一些类型，支持运行时类型发现与类型的动态创建
System.Runtime.InteropServices	这个命名空间提供了一些设施，使得.NET类型可以与“非托管代码”交互（例如，基于C的DLL和COM服务器），或反过来
System.Drawing System.Windows.Forms	这些命名空间定义了使用.NET原始UI工具包（Windows Forms）来构建桌面应用程序所用到的类型
System.Windows System.Windows.Controls System.Windows.Shapes	System.Windows命名空间是一些表示WPF UI工具包的几个命名空间的根
System.Linq System.Xml.Linq System.Data.DataSetExtensions	这些命名空间定义了针对LINQ API编程时用到的类型
System.Web	这个命名空间用来构建ASP.NET Web应用程序
System.ServiceModel	这个命名空间用来通过WCF API构建分布式应用程序
System.Workflow.Runtime System.Workflow.Activities	这两个命名空间定义了使用WWF API构建支持工作流的应用程序的类型
System.Threading System.Treading.Tasks	这个命名空间定义了可以用来构建多线程应用程序（将工作负载分配到多个CPU上）的类型
System.Security	安全是.NET中的一个不可分割的方面。在这个以安全为中心的命名空间中，有很多用来处理权限、加密等问题的类型
System.Xml	这个以XML为中心的命名空间包括了众多用于与XML数据交互的类型

1.8.1 Microsoft根命名空间的作用

在看表1-3时，你应该注意到了System是许多嵌套命名空间（如System.IO、System.Data等）的根命名空间。然而，其实.NET基础类库定义了许多System之外的最高层根命名空间，其中最有用的叫做Microsoft。

简而言之，任何Microsoft的嵌套命名空间（如Microsoft.CSharp、Microsoft.ManagementConsole以及Microsoft.Win32）包含的类型都用于和那些只属于Windows操作系统的服务进行交互。这样的话，我们可以认为这些类型不能在其他诸如Mac OS X等支持.NET的操作系统上运行。本书在大多数情况下不会深入到Microsoft根命名空间中的一些细节，因此如果你感兴趣的话，请务必查阅.NET Framework 4.5 SDK文档。

说明 第2章举例说明了.NET Framework 4.5 SDK文档的作用，它提供了能在基础类库中找到的每一个命名空间、类型和成员的细节。

1.8.2 以编程方式访问命名空间

命名空间只是一种方便我们从逻辑上理解和组织关联类型的方式，这一点应该反复强调。我们再来考虑System命名空间。从你的角度看，可以假设System.Console表示一个在System命名空间中名为Console的类，然而从.NET运行库的角度看，它却不是。运行时引擎只认识名为System.Console的独立实体。

在C#中，using关键字简化了引用特定命名空间中定义的类型的过程。为什么呢？假设要使用WF API建立一个图形化桌面应用程序，主窗口基于后台数据库信息呈现一个柱状图表并显示公司图标。理解每个命名空间包含的类型需要一定的学习和实践，下面是应用程序中经常引用的一些命名空间。

```
// 这里列出所有构建这个应用程序需要使用的命名空间
using System;                    // 通用基础类库类型
using System. Windows.Shapes;    // 图形呈现类型
using System.Windows.Controls;   // GUI窗口部件类型
using System.Data;               // 通用以数据为中心的类型
using System.Data.SqlClient;     // MS SQL Server数据访问类型
```

指定了若干命名空间（并设置好指向定义它们的程序集的引用）以后，就可以随意创建这些命名空间包含的类型的实例了。举例来说，如果想建立一个Button类（在System.Windows.Controls命名空间中定义）的实例，可以这样写：

```
// 显式列出这个文件所使用的命名空间
using System;
using System.Windows.Controls;

class MyGUIBuilder
{
  public void BuildUI()
  {
```

```
      // 创建按钮控件
      Button btnOK=new Button();
      ...
    }
  }
```

因为代码文件引用了System.Windows.Controls，所以编译器能够将Button类解析为这个命名空间的成员。如果没有特别指定System.Windows.Controls命名空间，将出现一个编译器错误。然而，还是可以使用完全限定名（full qualified name）[①]来声明变量：

```
// 不使用引入System.Windows.Controls的命名空间
using System;

class MyGUIBuilder
{
  public void BuildUI
  {
    // 使用完全限定名
    System.Windows Coutrols Button btnOK=
      new System.Windows.Controls.Button();
    ...
  }
}
```

虽然使用完全限定名定义一个类型可以提高程序的易读性，但C#的 using关键字能够减少按键次数。本书将选择C# using关键字的简化方式，而不使用完全限定名（除非它们的定义含糊不清，可能发生歧义）。

然而，请记住using关键字只是特定类型的完全限定名的简单速记符号，每种方法最后都会得出相同的底层CIL（事实上，CIL代码总是使用完全限定名），并且对程序集的大小和性能没有任何影响。

1.8.3 引用外部程序集

除了通过C# using关键字来指定命名空间，还需要告诉C#编译器包含引用类型的实际CIL定义的程序集的名字。如前面所提到的，许多核心的.NET命名空间包含在mscorlib.dll文件中。但System.Drawing.Bitmap类型包含在另一个名为System.Drawing.dll的程序集中。大多数.NET Framework程序集都位于称为全局程序集缓存（Global Assembly Cache，GAC）的特定目录下。在安装Windows的计算机上，全局程序集缓存默认状态下都位于C:\Windows\Assembly目录下，如图1-6所示。

根据构建.NET应用程序所用的开发工具的不同，可以有多种不同的方法来告知编译器在编译期间要包括哪些程序集。下一章将会看到如何实现，在此就不再赘述。

说明 从.NET 4.0开始，微软选择将.NET 4.0和.NET 4.5程序集隔离到一个特定的位置，使之与C:\Windows\ Assembly毫无关系。第14章将详细介绍这一点。

---

① 指带有完整命名空间的名字。——编者注

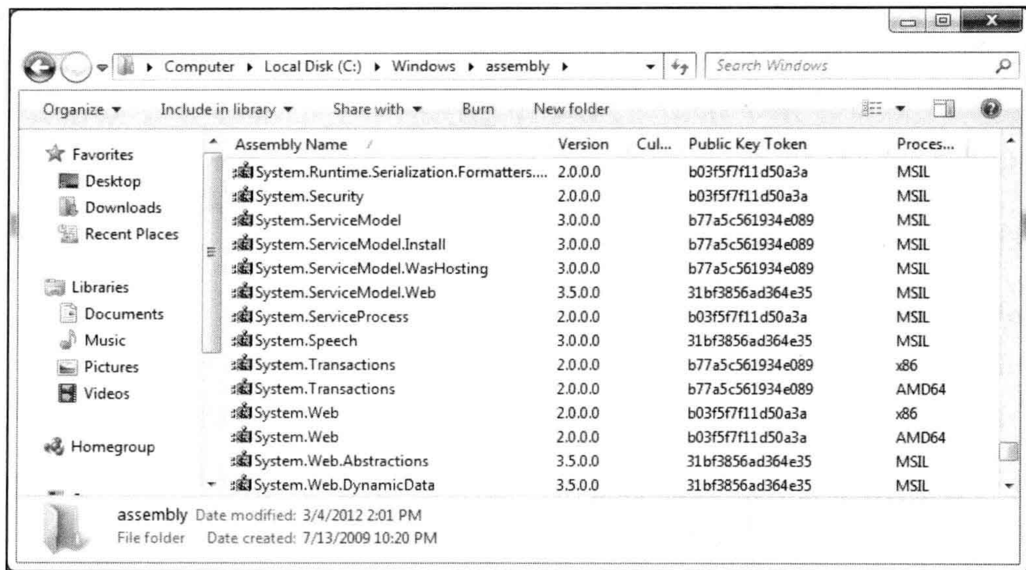


图1-6　很多.NET库都在全局程序集缓存中

1.9　使用 ildasm.exe 探索程序集

如果感到掌握.NET平台中的每一个命名空间有些困难，那么只需要记住，各命名空间的区别在于，它们包含了语义上具有一定关联的类型。所以，如果只需要一个控制台程序而不需要用户界面，就可以不需要了解桌面和Web命名空间。如果要建立一个绘画应用程序，那么很可能不会涉及数据库命名空间。同其他任何新的预制代码集类似，你可以边干边学。

与.NET Framework 4.5 SDK同时发布的中间语言反汇编工具（ildasm.exe）可以加载任意的.NET程序集并分析它的内容，包括关联的清单、CIL代码和类型元数据。这个工具允许程序员深入研究C#代码和CIL的对应关系，并会帮助我们理解.NET平台的内部工作原理。要成为一名优秀的.NET程序员，你不一定非得会用ildasm.exe，只需时不时地打开这个工具，琢磨一下C#代码和运行库概念的关系即可。

说明　打开Developer Command Prompt，输入“ildasm”（不含引号）并回车，可以方便地运行ildasm.exe。第2章会介绍如何打开这种特殊的命令窗口，以及从命令行窗口加载的其他工具。

运行这个工具后，打开File→Open菜单命令，选择一个想浏览的程序集。图1-7显示了前面所述的根据Calc.cs文件生成的名为Calc.exe的程序集，ildasm.exe使用我们熟悉的树状图展示程序集的结构。

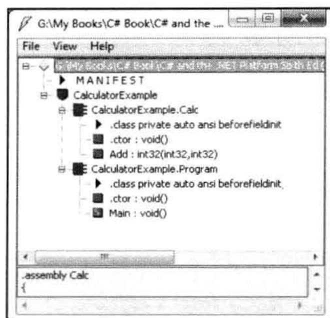


图1-7　通过ildasm.exe可以查看.NET程序集中的CIL代码、清单和元数据

1.9.1　查看 CIL 代码

除了显示指定程序集中包含的命名空间、类型和成员外，ildasm.exe还可以查看指定成员的CIL指令。例如，双击Program类的Main()方法，将弹出一个窗口显示底层CIL代码（见图1-8）。

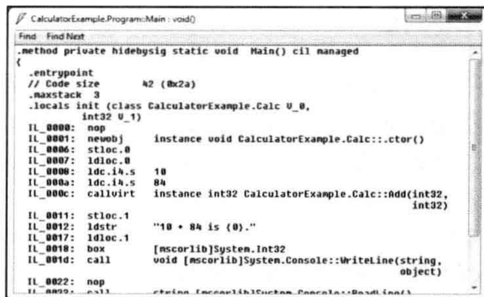


图1-8　查看底层CIL

1.9.2　查看类型元数据

如果想查看当前加载的程序集的类型元数据，按Ctrl+M组合键。图1-9显示了Calc.Add()方法的元数据。

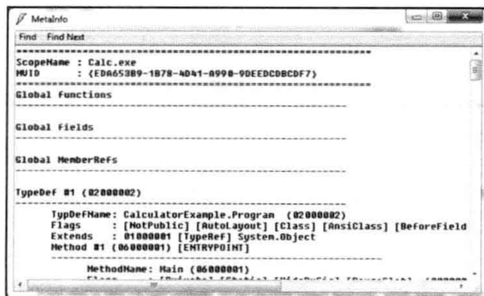


图1-9　通过ildasm.exe查看类型元数据

1.9.3　查看程序集元数据（即清单）

最后，如果想查看程序集清单的内容，只需双击MANIFEST图标（如图1-10所示）。

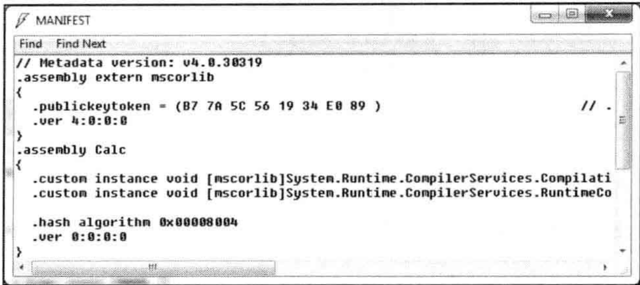


图1-10　通过ildasm.exe查看清单数据

当然，ildasm.exe还有更多的选项，我将在适当的地方举例说明这个工具的其他功能。

1.10　.NET 的平台无关性

接下来简单说一下.NET平台的平台无关性。令许多程序员惊讶的是，.NET程序集可以在非微软操作系统[如Mac OS X、各种版本的Linux、Solaris，以及iOS和Android移动设备（通过Mono Touch API）上开发和执行。要理解它是怎么做到的，需要掌握.NET领域中的另外一个缩写词CLI（Common Language Infrastructure，公共语言基础设施）。

当微软发布C#语言和.NET平台时，也发布了一整套正式的文档来说明C#和CIL语言的语法及语义、.NET程序集格式、核心.NET命名空间以及假定的.NET运行时引擎的结构（叫做虚拟执行系统，即VES）。

让我们高兴的是，这些文档已经提交到ECMA，并被ECMA批准成为官方的国际标准（http://www.ecma-international.org）。我们感兴趣的规范主要是：

- ECMA-334，C#语言规范；
- ECMA-335，公共语言基础设施（CLI）。

它们可以使第三方组织在各种操作系统和处理器上构造不同的.NET平台发行版，理解了这一点，这些文档的重要性就显而易见了。ECMA-335也许是这两个规范中更“有内容”的一个，它分为六部分，如表1-4所示。

表1-4　CLI的各个部分

ECMA-335的各个部分	含　义
部分I：概念架构	描述了整个CLI的架构，包括CTS和CLS的规则以及.NET运行时引擎的机制
部分II：元数据的定义和语义	描述了.NET元数据和程序集格式的细节
部分III：CIL指令集	描述了CIL代码的语法和语义
部分IV：配置文件和库	在较高层次概述了必须由.NET发行版支持的最少的和最完整的类库
部分V：调试交换格式	描述了CLI生产商和消费者之间交换调试信息的标准方式
部分VI：附录	其他内容，如关于类库设计指南和CIL编译器实现的细节

需要注意，部分IV（配置文件和库）仅仅定义了最基本的命名空间集，这些命名空间表示一个CIL发行版应有的核心服务（如集合、控制台I/O、文件I/O、线程、反射、网络访问、核心安全需求、XML数据处理等）。CLI没有定义用于Web开发（ASP.NET）、数据库访问（ADO.NET）或桌面图形用户界面（GUI）应用开发（Windows Forms/WPF）的命名空间。

令人高兴的是，为了提供具有完整功能的、产品级的开发平台，主流的.NET分发版都扩展了CLI库，加入了与微软ASP.NET、ADO.NET和Windows Forms兼容的等效功能。目前，CLI实现有两个主要的流派（除了微软提供的针对Windows的实现之外）。虽然本书专注于使用微软.NET编写.NET应用程序，但表1-5也提供了Mono和Portable.NET项目的相关信息。

表1-5 开源.NET分发版

发行版	作用
www.mono-project.com	Mono项目是一个CLI的开源分发版，针对各种Linux分发版（如SUSE、Fedora等）以及Windows和Mac OS X /iOS设备（iPad、iPhone）设备
www.dotgnu.org	Portable.NET是另外一个CLI的开源分发版，运行在很多操作系统上。Portable.NET的目的是在尽可能多的操作系统上运行（如Windows、AIX、Mac OS X、Solaris，以及所有主要的Linux发行版本）

Mono和Portable.NET都提供符合ECMA标准的C#编译器、.NET运行时引擎、代码示例、说明文档和许多与微软的.NET Framework 4.5 SDK功能相当的开发工具。

说明 附录A介绍了如何使用Mono创建跨平台的.NET应用程序。

1.11 Windows 8应用程序简介

本章最后将简述微软Windows 8操作系统推出的全新技术与.NET平台有何关联。如果你还没时间探索Windows 8，我只想告诉你，全新的用户界面与之前所有的微软Windows版本都不相同（事实上，我想说的是，对于还有印象的老用户来说，这次变革就像Windows 3.11到Windows 95那次一样大）。图1-11展示了Windows 8操作系统全新的开始界面。

如图1-11所示，Windows 8开始界面支持平铺布局，每个方块都表示机器上安装的一个应用程序。与典型的Windows桌面应用不同，Windows 8应用程序专为触摸屏交互而构建。它们还都以全屏运行，没有平常在很多桌面应用中都有的那些“金属块”（如菜单系统、状态条、工具条按钮）。图1-12展示了一个Windows 8天气应用的部分视图。

说明 构建Windows 8应用程序要求开发者遵循一套全新的UI设计原则、数据存储方法和用户输入选项。没错，一个“Win8”应用远不止一个开始界面上的方块那么简单。



图1-11　Windows 8开始界面

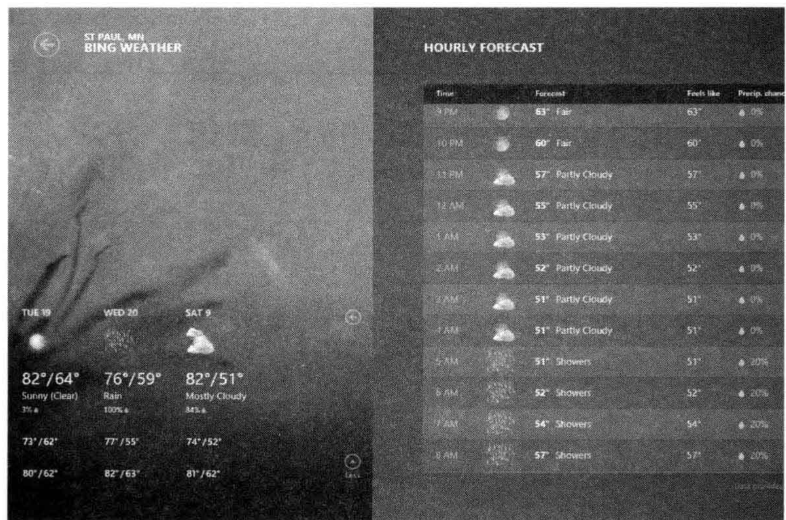


图1-12　Windows 8应用是全屏的、富图形化的桌面应用

1.11.1　构建Windows 8 应用程序

创建和运行Windows 8应用程序只能在Windows 8上，而不能在Windows 7上。实际上，如果在Windows 7（或更早的版本）上安装Visual Studio，在New Project对话框中根本就看不到Windows 8项目模板。

编写Windows 8应用程序要求开发者深入了解一个全新的运行库层——Windows Runtime（WinRT）。要清楚的是，WinRT不是.NET CLR，尽管它提供了一些相似的服务，如垃圾收集、多编程语言支持（包括C#）等。

此外，这些应用程序使用全新的命名空间创建，所有这些命名空间都以根名Windows开头。图1-13在Visual Studio对象浏览器中展示了不同的WinRT命名空间。

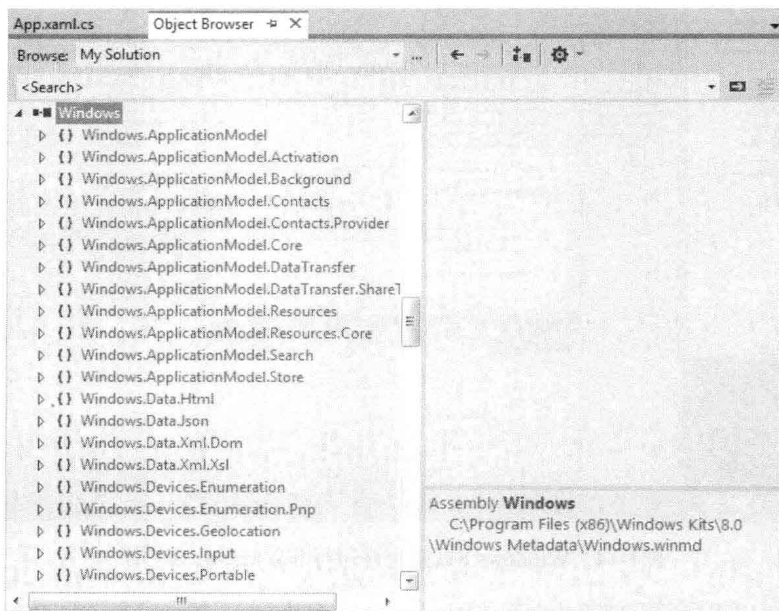


图1-13 `Windows.*`命名空间只能用于构建Windows 8应用程序

幸运的是，`Windows.*`命名空间提供的功能反应了很多.NET基类库的API。实际上，从编程的视角来看，为WinRT构建应用程序与为CLR构建.NET应用程序十分类似。例如，Windows 8应用程序可用C#（以及Visual Basic、JavaScript或C++）构建。

并且，很多`Windows.*`命名空间提供的功能与.NET基础类库中的相似。例如，本书后面（参见第27章至第31章）将介绍的一项.NET技术WPF。届时你将学习一种基于XML的语言XAML。当你开始探索Windows 8应用程序的时候，你会很高兴地发现，它们也是用XAML构建的（或者如果你选择的语言是JavaScript，将使用HTML5），并且和在.NET平台构建的WPF程序具有十分相似的编程模型。

1.11.2 .NET在Windows 8 中的作用

除了`Windows.*`命名空间，Windows 8应用程序还能使用大量.NET平台的子集。总的来说，Windows 8应用能使用的.NET API支持泛型集合、LINQ、XML文档处理、I/O服务、安全服务以及本书将要介绍的其他特性。这当然是一件好事，因为本书介绍的很多话题可以直接映射到Windows 8应用程序的构造上。

还要注意的是，尽管在Windows 8下默认显示的不是经典的Windows桌面，但你仍然可以通过点击Desktop方块或简单地按下键盘上的Windows键来访问。一旦打开，操作系统将显示与Windows 7桌面看上去完全相同的桌面（如图1-14所示）。

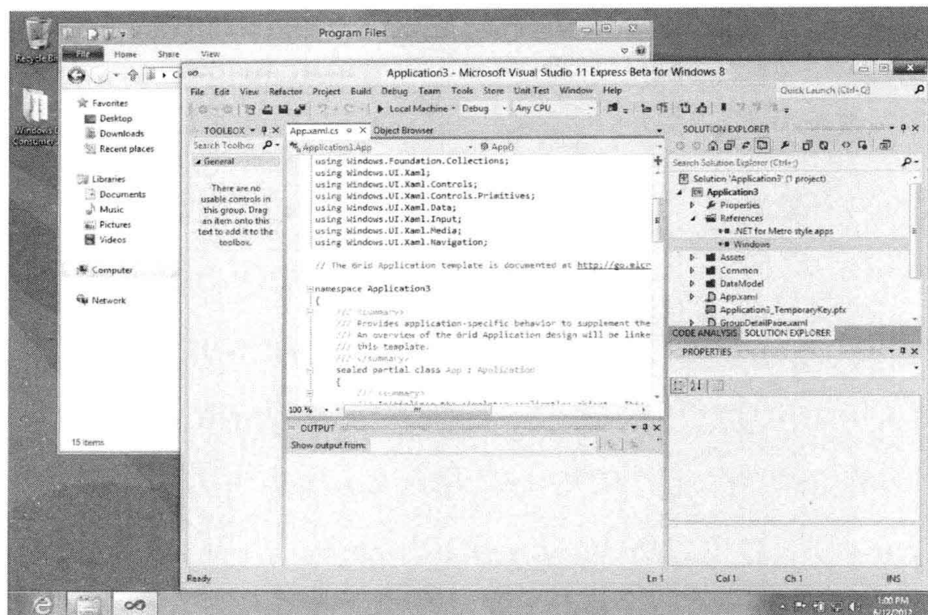


图1-14　Windows 8桌面允许运行非Windows 8应用

在这里，你可以运行所有期望的桌面应用，包括Visual Studio、Word、Excel、Photoshop等。同样，如果在Windows 8下构建.NET应用程序，将运行在Windows 8桌面环境中。

因此简单地说，微软.NET平台在Windows 8下照样可以使用，而且将是你开发工作中的重要部分。只要记住，.NET本身并不是用来构建Windows 8应用程序的。相反，你需要使用新库（Windows.*）、新运行库（WinRT）和.NET子集（.NET API）来创建这样的程序。

本书不会探讨使用WinRT构建Windows 8应用程序的过程，而是会专注于.NET平台。尽管如此，要记住，本书介绍的大多数话题都会为你以后探索Windows 8应用开发做好充分的准备，你应该会有这个需求。

1.12　小结

本章的目的在于为本书其余部分建立起一个概念性的框架，由.NET之前各种技术的局限性和复杂性谈起，然后综述了.NET和C#是如何给出简化解决方案的。

.NET本质上就是一个运行库执行引擎（mscoree.dll）和基础类库（mscorlib.dll等）。CLR可以承载任何符合托管代码规则的.NET二进制文件（又称程序集）。而程序集中有很多CIL指令（以及类型元数据和程序集清单），这些指令通过即时编译器编译为特定平台的指令。另外，本章还讲述了CLS和CTS的作用。

接下来我们介绍了ildasm.exe和reflector.exe对象浏览工具的功能，以及怎样使用完整的客户配置文件来配置计算机，使之可以运行.NET应用程序。最后简要说明了C#和.NET的平台无关性（附录A会深入探讨这个主题），以及.NET平台在Windows 8操作系统中是如何定位的。

第2章

构建C#应用程序



作为一名C#程序员，构建.NET应用程序有许多工具可以选择。本章将介绍各种.NET开发工具，其中当然包括Visual Studio。本章先探讨如何使用C#命令行编译器csc.exe、微软Windows操作系统上最简单的文本编辑器——记事本（Notepad）应用程序以及可以免费下载的Notepad++。

虽然在阅读本书的过程中可以只使用csc.exe和基本的文本编辑器，我相信你肯定会对使用具有丰富功能的IDE感兴趣。为此，本书也会介绍名为SharpDevelop的免费的开源IDE。这种IDE可与许多商用的.NET开发环境相媲美。接下来简单介绍Visual C# Express IDE（这也是免费的），然后我们将把注意力转向Visual Studio Professional的主要功能。

说明 本章有大量的C#语法，之前我们还未正式讨论过。如果对语法不熟悉，不用烦恼，因为第3章会正式探讨C#语言。

2.1 .NET Framework 4.5 SDK 的作用

有关.NET开发常见的一个误解就是，程序员必须购买Visual Studio才能构建C#应用程序。事实上，使用可免费下载的.NET Framework 4.5 Software Development Kit（SDK）就可以构建任何形式的.NET程序。

SDK提供了很多托管的编译器、命令行工具、示例代码、.NET类库以及完整的文档系统。如果使用Visual Studio或Visual C# Express，就没有必要手动安装.NET Framework 4.5 SDK。当我们安装其中任一产品的时候，都会自动安装SDK，所有的东西都是现成的。然而，如果你不准备使用微软IDE来学习本书的话，请务必在继续之前安装SDK。

说明 .NET Framework 4.5 SDK安装程序（dotNetFx45_Full_x86_x64.exe）可以从.NET网站下载（http://msdn.microsoft.com/netframework）。

开发者命令提示符

如果安装.NET Framework 4.5 SDK、Visual Studio或Visual C# Express，最后会在本地硬盘上多出

很多新的目录，每一个目录都包含各种.NET开发工具。其中很多工具都需要从命令提示符打开，因此，如果希望在任何Windows命令窗口中使用这些工具，就需要为操作系统注册路径。

虽然你可以手动更新PATH变量，不过也可以使用Start→All Programs→Microsoft Visual Studio 11→Visual Studio Tools文件夹的Developer Command Prompt来节省你的时间（如图2-1所示）。

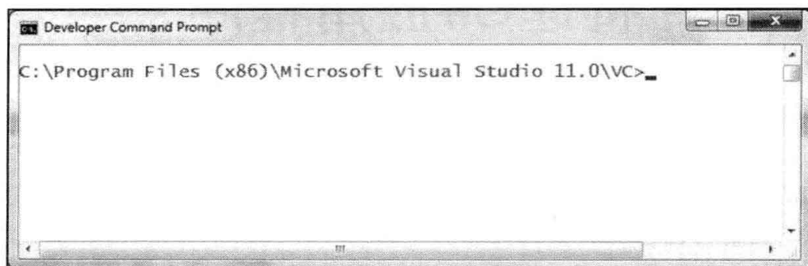


图2-1　The Developer Command Prompt

使用特定的命令提示符的好处在于它已经预配置了每一个.NET开发工具。如果已经安装了.NET开发环境，只需输入如下命令然后按Enter键：

```
csc -?
```

如果一切正常的话，应该可以看到C#命令行编译器（csc代表C-sharp compiler）的命令行参数列表。如你所见，该命令行编译器有不少选项；但实际上在命令行提示符中编写C#程序只需要其中几个设置。

2.2　用 csc.exe 构建 C#应用程序

尽管你可能从来不用C#命令行编译器来生成大型的应用程序，但理解如何亲手编译自己的代码文件的基本知识还是很重要的。为什么应该掌握这一过程呢？以下是几个原因。

- 最显而易见的原因是一个简单的事实：你可能没有Visual Studio 或者另一个图形化IDE。
- 你可能在大学里，那里会禁止在教室里使用代码生成工具或IDE。
- 你想要使用自动的构建工具，如msbuild.exe，这就需要你了解你正使用的工具的命令行选项。
- 你想要加深对C#的理解。使用图形化IDE来构建应用程序时，最终是在指导csc.exe如何操纵C#导入文件。这样你就可以明了在后台到底发生了什么。

使用原始csc.exe的另一个好处是，你将更熟悉对.NET Framework 4.5 SDK所包含的其他命令行工具的操作。在阅读本书的过程中你会发现，一些重要的工具只能够通过命令行方式访问（如gacutil.exe、ngen.exe、 ilasm.exe和 aspnet_regiis.exe）。

为了说明在不使用IDE的情况下如何构建.NET应用程序，我们使用C#命令行编译器和记事本生成名为TestApp.exe的一个简单的可执行程序集。首先，需要一些源代码。打开记事本（通过Start→All Programs→Accessories菜单选项）并键入以下内容：

```
// 一个简单的C#应用程序
using System;
```

```
class TestApp
{
  static void Main()
  {
    Console.WriteLine("Testing! 1, 2, 3");
  }
}
```

完成后把文件以TestApp.cs的名字保存在一个方便的地方（例如C:\CscExample）。现在，我们来了解C#编译器的核心选项。

说明 根据惯例，所有C#代码的文件扩展名都是*.cs。文件的名字不需要有任何指向类型名定义的映射。

2.2.1 指定输入输出目标

首先要明白如何指定要创建的程序集的名字和类型（例如，控制台应用程序命名为MyShell.exe、代码库命名为MathLib.dll、WPF应用程序命名为Halo8.exe，等等）。可以通过将对应的具体标志作为命令行参数传入csc.exe来选择各种选项（如表2-1所示）。

表2-1 C#编译器的输出选项

选　项	作　用
/out	本选项用于指定将被构建的程序集的名字。默认条件下，程序集的名字与最初输入的*.cs文件名字相同
/target:exe	本选项构建一个可执行的控制台应用程序。这是默认的程序集输出类型，并且在创建该应用程序类型时可被忽略
/target:library	本选项构建一个文件*.dll程序集
/target:winexe	尽管使用/target:exe选项也能创建基于GUI的应用程序，但本选项创建的程序运行时不会有控制窗口出现在桌面背景上

说明 发送到命令行编译器（以及其他大多数命令行工具）的选项可以以短横线（-）或斜线（/）为前缀。

为了把TestApp.cs编译成名为TestApp.exe的控制台应用程序，使用cd（change directory）命令转到包含源代码文件的目录

```
cd C:\CscExample
```

并键入以下命令（注意命令行标志必须位于导入的文件名字前面，不能在后面）：

```
csc /target:exe TestApp.cs
```

这里没有明确指定/out标志，因而如果TestApp是传入文件的名字，可执行文件将被命名为TestApp.exe。还要清楚的是，大多数C#编译器标志支持缩写版本，例如可以用/t代替/target（程序员可以在命令提示符下键入csc -?来查看所有的缩写）：

```
csc /t:exe TestApp.cs
```

而且，因为/t:exe标志是C#编译器的默认输出，也可以只键入下面的命令来编译TestApp.cs：

```
csc TestApp.cs
```

现在TestApp.exe可以从命令行运行了，如图2-2所示。

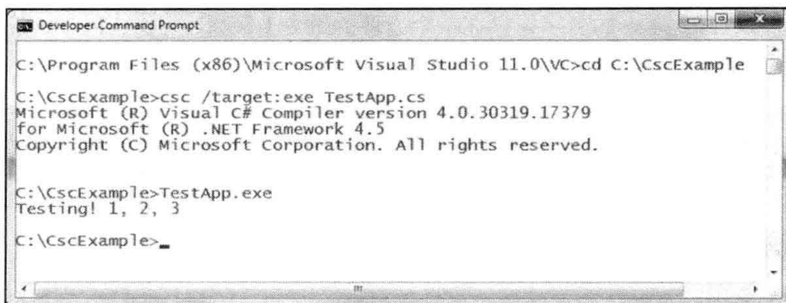


图2-2　TestApp.exe在运行

2.2.2　引用外部程序集

接下来，看一下如何编译一个应用程序，如果它采用了在另一个.NET程序集里定义的类型。说到这里，如果你想知道C#编译器是如何理解对于System.Console类型的引用的，请回忆一下第1章里mscorlib.dll在编译过程期间是如何被自动引用的（如果由于特别的原因希望禁用这个功能，可以指定csc.exe的/nostdlib选项）。

让我们修改TestApp应用程序，显示一个Windows窗体消息框。打开TestApp.cs文件并做如下修改：

```
using System;

// 一定要加上这一行
using System.Windows.Forms;

class TestApp
{
  static void Main()
  {
    Console.WriteLine("Testing! 1, 2, 3");

    // 一定要加上这一行
    MessageBox.Show("Hello...");
  }
}
```

注意，对System.Windows.Forms命名空间的引用是通过C# using关键字实现的（对该关键字的介绍见第1章）。回忆一下，如果显式地列出在一个给定的*.cs文件里所用到的命名空间，就可以避免采用完全限定名（否则你会累到手抽筋）。

在命令行中，必须通知csc.exe，哪个程序集包含了“正在使用的”命名空间。假定你已经使用了System.Windows.Forms.MessageBox类，就必须使用/reference（可以缩写为/r）标志指定System.Windows.Forms.dll程序集：

```
csc /r:System.Windows.Forms.dll TestApp.cs
```

如果现在再次运行应用程序，除了控制台输出以外，还可以看见图2-3中出现的消息框。

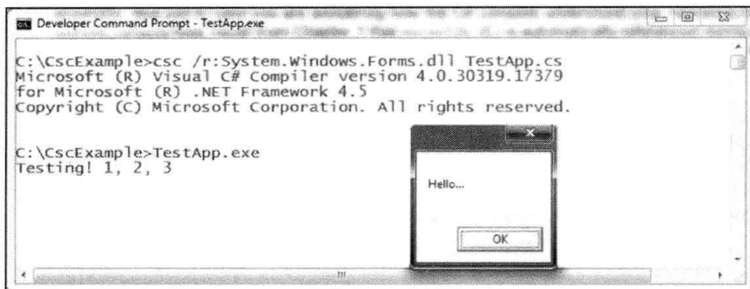


图2-3　第一个图形化用户界面应用

2.2.3　引用多个外部程序集

顺便说明，如果需要csc.exe引用大量的外部程序集，会怎样呢？仅需使用一个用分号分隔的列表列出各个程序集。对上面这个例子不需要指定多个外部程序集，但我们还是给出一个示例用法：

```
csc /r:System.Windows.Forms.dll;System.Drawing.dll *.cs
```

说明　正如本章稍后将要介绍的那样，即使没有使用/r标志进行指定，C#编译器也将自动引用一些.NET核心程序集（如System.Windows.Forms.dll）。

2.2.4　编译多个源文件

TestApp.exe应用程序的当前版本是用单个*.cs源代码文件创建的。在单个*.cs文件里定义所有.NET类型是完全允许的，然而大多数项目由多个*.cs文件组成，以使代码库更灵活。假设你又编写了一个类，包含在名为HelloMsg.cs的新文件里：

```
// HelloMessage类
using System;
using System.Windows.Forms;

class HelloMessage
{
  public void Speak()
  {
    MessageBox.Show("Hello...");
  }
}
```

现在，修改初始的TestApp类以使用这种新的类类型，并且注释掉以前的Windows窗体代码：

```
using System;

// 不再需要这一行了
```

```
// using System.Windows.Forms;

class TestApp
{
  static void Main()
  {
    Console.WriteLine("Testing! 1, 2, 3");

    // 也不再需要这一行了
    // MessageBox.Show("Hello...");

    // 使用HelloMessage类
    HelloMessage h = new HelloMessage();
    h.Speak();
  }
}
```

可以通过显式地列出各个导入文件来编译C#文件：

```
csc /r:System.Windows.Forms.dll TestApp.cs HelloMsg.cs
```

另外，C#编译器还允许使用通配符（*）通知csc.exe，将所有位于项目目录里的*.cs文件作为当前构建的一部分。

```
csc /r:System.Windows.Forms.dll *.cs
```

当再次运行程序时，输出与前面的代码完全相同。两个应用程序的唯一差别在于，当前代码分到了多个文件中。

2.2.5　使用C#响应文件

可以想象，如果要在命令提示符下构建一个复杂的C#应用程序，那么将不得不指定大量的输入选项以通知编译器如何处理源代码。为了减轻录入负担，C#编译器采用了响应文件（response file）。

C#响应文件包含了在当前程序的编译期间要用到的所有指令。通常约定，这些文件的扩展名为*.rsp。假定已经创建了一个包含有以下选项的名为TestApp.rsp的响应文件（可以看到，注释用#字符标识）：

```
# 这是第2章里的TestApp.exe示例的响应文件

# 外部程序集引用
/r:System.Windows.Forms.dll

# 用于编译的输出和文件（采用通配符语法）
/target:exe /out:TestApp.exe *.cs
```

现在，假定该文件与将被编译的C#源代码文件保存在相同的目录里，这样就能按照以下步骤构建完整的应用程序了（注意采用了@符号）：

```
csc @TestApp.rsp
```

如果需要，也可以指定多个*.rsp文件作为输入（例如csc @FirstFile.rsp @SecondFile.rsp @ThirdFile.rsp）。如果采用这种方式，要记住编译器会根据所遇到的命令选项做相应的处理！因而，后面*.rsp文件中的命令行参数可以覆盖前一个响应文件的选项。

还要注意，在响应文件前的命令行中被显式列出来的标志将被指定的*.rsp文件覆盖，因此，如果键入：

```
csc /out:MyCoolApp.exe @TestApp.rsp
```

假定有在TestApp.rsp响应文件里列出的/out:TestApp.exe标志，程序集的名字将仍然是TestApp.exe而非MyCoolApp.exe。但是，如果在响应文件后列出了标志，标志将覆盖响应文件里的设置。

说明 /reference标志具有累加性。不管在何处指定了外部程序集（之前、之后或是在响应文件内），最终的结果都是各个引用程序集的累加之和。

默认的响应文件（csc.rsp）

关于响应文件，最后要说明一点，C#编译器有一个与之关联的默认响应文件（csc.rsp），该默认响应文件与csc.exe同处在一个目录里（默认为C:\Windows\Microsoft.NET\Framework\<version>，其中<version>是给定平台的版本号）。如果用记事本打开这个文件，将发现无数的.NET程序集已经使用/r:标志被指定，包括Web开发用到的各种库、LINQ、数据访问和其他的核心库（除了mscorlib.dll以外）。

当用csc.exe构建C#程序时，即使你提供了一个自定义的*.rsp文件，该响应文件也将自动被引用。假定有默认的响应文件存在，当前的TestApp.exe应用程序可以用以下命令集成功地进行编译（System.Windows.Forms.dll在csc.rsp内被引用）：

```
csc /out:TestApp.exe *.cs
```

如果希望取消自动读取csc.rsp，可以指定/noconfig选项：

```
csc @TestApp.rsp /noconfig
```

说明 如果引用了从未用过的程序集（通过/r选项），它们将会被编译器忽略，因此，不用担心“代码膨胀”。

很明显，C#命令行编译器还有很多其他选项，用于控制如何产生结果.NET程序集。你将在本书相应的地方看到其他重要的特性，但更详细的内容可以参阅.NET Framework 4.5 SDK文档。

源代码 CscExample应用程序可在Chapter 2子目录中找到。

2.3 使用 Notepad++构建.NET 应用程序

我想提到的另一个文本编辑器是开源的Notepad++应用程序。这个工具可以从http://notepad-plus.sourceforge.net获得。与简单的Windows Notepad程序不同，Notepad++允许编写各种语言的代码以及安装各种插件。此外，Notepad++还提供了很多其他好东西，如：

- 对C#关键字（包括关键字颜色编码）的原生支持；
- 支持语法折叠，即允许折叠和展开编辑器中的一组代码行（和Visual Studio/C# Express相似）；
- 能通过Ctrl键和鼠标滚轮放大/缩小文本；
- 对各种C#关键字以及.NET命名空间进行可配置自动完成。

有关最后一点，按Ctrl+空格组合键，会激活C#的自动完成支持（如图2-4所示）。

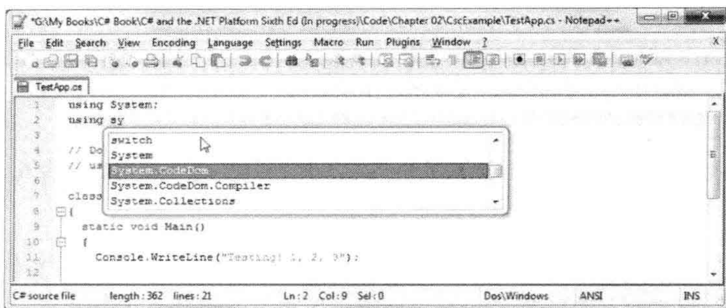


图2-4 使用Notepad++执行自动完成

说明 自动完成窗口内显示的选项列表可以被修改或者扩展。只需要打开C:\Program Files (x86)\Notepad++\plugins\APIs\cs.xml文件，编辑并且增加其他项即可。

除了现在介绍的内容外，我不会深入介绍Notepad++的其他细节。如果你需要更多协助的话，可以选择?Help菜单选项。

2.4 使用 SharpDevelop 构建.NET 应用程序

使用Notepad++编写C#代码相对于Notepad来说是一个进步。然而，这些工具没有为C#代码、构建图形用户界面的设计人员、项目模板或数据库操作工具提供丰富的智能感知能力。为了满足这个需求，本节将介绍下一个.NET开发的选项：SharpDevelop（也叫#Develop）。

SharpDevelop是一种功能丰富的开源IDE，通过它可以用C#、VB、IronRuby、 IronPython、C++、F#和类似Python的.NET语言Boo来构建.NET程序集。这种IDE是完全免费的，并且是完全用C#编写的。实际上，你可以下载并手动编译*.cs文件，或运行setup.exe程序，从而在开发计算机上安装SharpDevelop。以上两种方式在网址http://www.sharpdevelop.com处都可以找到相应的下载文件。

SharpDevelop提供了许多提高工作效率的手段。下面是其主要的优点：

- 支持多种.NET语言、.NET版本和项目类型；
- 具有智能感知、代码自动完成和插入代码段的能力；
- 具有Add Reference对话框，用于引用外部程序集，包括部署到全局程序集缓存的程序集；
- 针对桌面和Web应用集成的GUI设计器；
- 具有一个集成的对象浏览和代码定义的工具；
- 具有可视的数据库设计器工具；
- 具有使C#与VB代码相互转换的工具。

作为免费的IDE，这已经很出色了，是不是？本章不可能详细地介绍这些特性，我们来讨论其中有趣的几个。

说明 在撰写本书的时候，SharpDevelop的最新版本（4.2）还不支持C#/.NET 4.5的特性。请检查SharpDevelop网站检查最新版本。

构建简单的测试项目

安装了SharpDevelop之后，通过File→New→Solution菜单选项就可以选择待生成的项目类型和.NET语言的类型。例如，假定新建一个C# Windows应用程序MySDWinApp（如图2-5所示）。

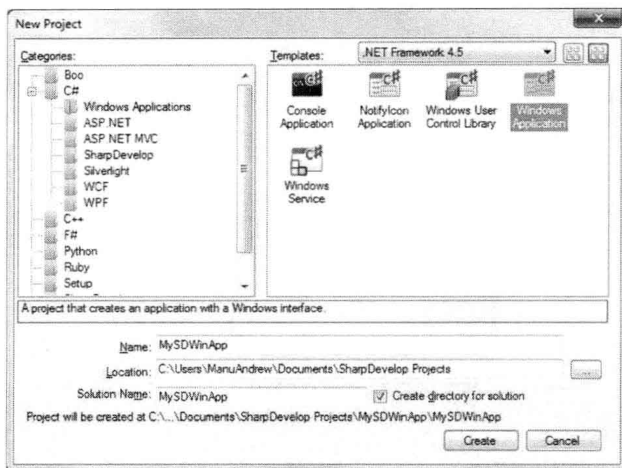


图2-5　SharpDevelop的New Project对话框

和Visual Studio一样，这里有一个Windows Forms GUI设计器工具箱（用来把控件拖放到设计器上）和一个Properties窗口，用来设置每个UI项的外观。图2-6描述了如何使用IDE配置一个按钮控件（注意我单击了打开的代码文件底部的Design标签）。

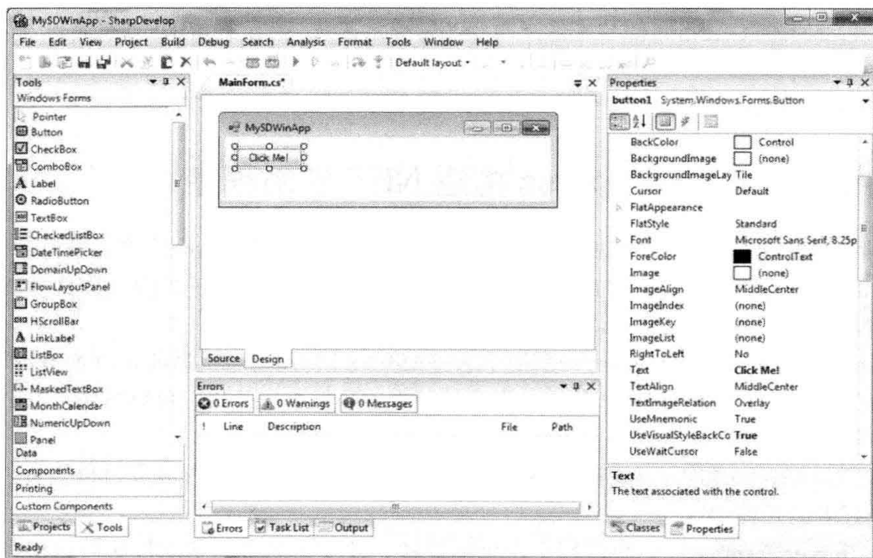


图2-6　使用SharpDevelop以图形方式设计Windows Forms应用程序

单击表单设计器底部的Source按钮，将会发现期望的智能感知、代码完成以及集成的帮助功能（如图2-7所示）。

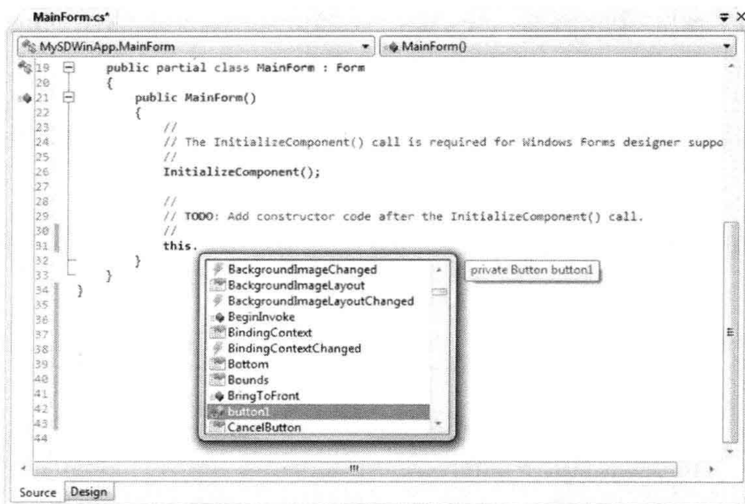


图2-7　SharpDevelop支持许多代码生成工具

SharpDevelop的设计在很多功能上酷似微软的.NET IDE（我们后面会讨论）。因此，我不会深入探讨这个开源.NET IDE的所有特性。如果你需要了解更多信息的话，请使用Help菜单。

说明　MonoDevelop是基于Sharp Develop 代码的开源IDE。MonoDevelop 是.NET程序员使用Mono平台创建Mac OS X或Linux 操作系统应用程序的首选IDE。要了解该IDE的更多情况请访问http://monodevelop.com/。

2.5　使用 Visual C# Express 构建.NET 应用程序

在2004年夏天，微软引入了一系列全新的IDE，它们被分类在"Express"产品系列下（http://msdn.microsoft.com/express）。目前，在"Express"产品系列里有大量成员（它们是完全免费的，并且被微软公司支持和维护），具体如下所示。

- Visual Web Developer Express：一个用于通过ASP.NET构建动态网站和WCF服务的轻量级工具。
- Visual Basic Express：一种非常适合程序员新手的高效编程工具，可以采用Visual Basic这种易于使用的语法构建应用程序。
- Visual C# Express和Visual C++ Express：针对选用不同的语法学习计算机科学基础知识的学生和爱好者的几种IDE。
- SQL Server Express：一个适用于业余爱好者、技术狂热者和学生程序员的初级数据库管理系统。

Visual C# Express的独特功能

概括地说，“Express”产品是同类型的Visual Studio产品的简化版本，并且主要针对.NET业余爱好者和学生。像SharpDevelop一样，Visual C# Express配备了各种对象浏览工具、桌面应用程序GUI设计器、Add References对话框、智能感知能力以及代码扩展模板。

此外，Visual C# Express提供了一些重要的特性，这些特性目前在SharpDevelop里还没有，包括：

- 对WPF XAML应用程序的丰富支持；
- 下载一些免费模板的功能，如支持Xbox 360开发或与Twitter集成的WPF应用程序的模板，等等。

由于Visual C# Express的外观界面与Visual Studio非常相似（并且在某种程度上也与SharpDevelop相似），在这里就不对这个具体的IDE进行阐述了。在阅读本书的过程中，可以随意地使用该IDE，但要知道Visual C# Express不支持构建ASP.NET网站的项目模板。如果你希望构建Web应用程序，同样可以在http://msdn.microsoft.com/express上下载Visual Web Developer。

2.6 使用 Visual Studio 构建.NET 应用程序

假如你是一位职业.NET软件工程师，很有可能公司会购买微软最重要的IDE——Visual Studio（http://msdn.microsoft.com/vstudio）。这个工具就是本章中功能最丰富、适用于企业的IDE。当然，一分钱一分货，价格随着所购买的Visual Studio的版本不同而不同。你可能会不信，但是各个版本都具有不同的功能。

说明 Visual Studio家族中有数量惊人的成员。在本书余下的部分中，我假设你选择了Visual Studio Professional作为IDE。

假定你选中的是Visual Studio Professional，但要知道，这对学习本书并不是必需的。在最坏的情况下，本书会提及你使用的IDE（比如Microsoft C# Express 或SharpDevelop）不支持的某个选项。不过请放心，本书的全部示例代码在被你所选用的工具进行处理时都可以正常编译。

说明 从Apress网址（http://www.apress.com）的Source Code/Downloads区域下载了本书的源代码后，你可以通过双击示例的*.sln文件将当前示例加载到Visual Studio或C# Express里。如果没有使用Visual Studio或者C# Express，你需要手动将所提供的*.cs文件插入到IDE的项目工作区中。

2.6.1 Visual Studio的独特功能

Visual Studio配备了我们所期望的GUI设计器、代码段支持、数据库操作工具、对象和项目浏览工具以及集成的帮助系统。与我们提过的那些IDE不同，Visual Studio提供了众多的附加功能。下面是其中的一部分：

- Visual XML编辑器/设计器；
- 支持Windows移动设备开发；
- 支持微软Office开发；

- 支持WWF项目的设计器；
- 对代码重构的集成支持；
- 可视类设计工具。

坦白地讲，Visual Studio提供了如此多的功能，以至于需要一整本书来完整描述IDE的各个方面，但这不是本书的目标。然而，我将在后面几页中给出Visual Studio的一些主要功能，你可以了解到Visual Studio IDE的更多内容。

说明　如果你使用过微软Visual Studio旧版本，会很快注意到整个外观和感觉与以前大不一样。如果你觉得默认的“Dark”主题太暗，可以选择Tools→Options菜单，在Environment→General部分选择Color Theme下拉列表，将主题改为“Light”。本书所有的截屏都使用的是“Light”主题。

2.6.2　使用New Project对话框指向.NET Framework

使用File→New→Project菜单项来新建一个C#控制台应用程序VsExample。如图2-8所示，Visual Studio支持选择你希望构建的.NET Framework的版本（2.0、3.0、3.5、4.0和4.5），可以使用New Project对话框顶部中心位置的下拉列表来选择。对于本书中的任何项目，你都可以使用默认的.NET Framework 4.5选项。

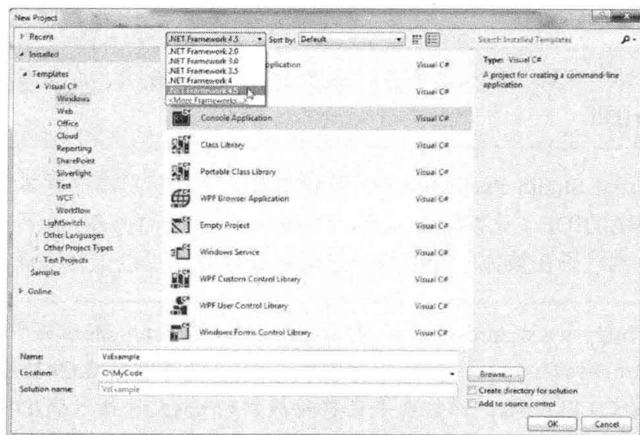


图2-8　Visual Studio允许我们指定某个版本的.NET Framework

2.6.3　解决方案资源管理器

解决方案资源管理器工具可以从View菜单得到，它可以用来查看构成当前项目的所有文件和被引用的程序集的集合（如图2-9所示）。此外，可以打开给定文件（比如Program.cs），查看当前文件定义的代码类型，后面，必要时我还会指出解决方案资源管理器的其他实用特性，但是你自己可以随意查看各个选项，看看它们各有什么作用。

注意，解决方案资源管理器的References文件夹显示了每一个你已经引用的程序集，根据你的项

目类型以及你编译针对的框架版本，程序集会有所不同。因为已经创建了一个Console Application，你会发现此处自动包含了一组必需的库（比如System.dll、System.Core.dll、System.Data.dll）。

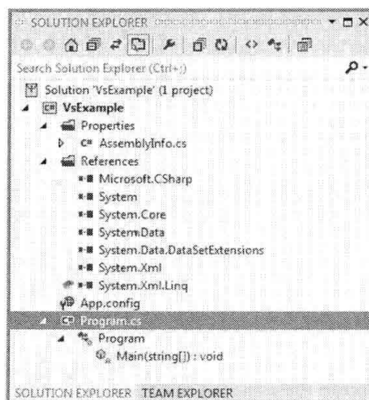


图2-9 解决方案资源管理器

说明 还记得第1章我们创建的第一个.NET库是mxcprlib.dll吗？这个库没有显示在解决方案资源管理器中，但是你可以使用其中包含的所有类型。

1. 引用外部程序集

当需要引用其他程序集时，右击References文件夹并且选择Add Reference。然后，你就可以从结果对话框中选择程序集了（本质上这是Visual Studio指定了命令行编译器的/reference选项）。Framework标签（如图2-10所示）显示了很多常用的.NET程序集，然而，Browse标签允许我们浏览本地磁盘上的任何.NET程序集。同样，Recent标签也很有用，它会保留几个你在其他项目中经常用到的引用程序集。

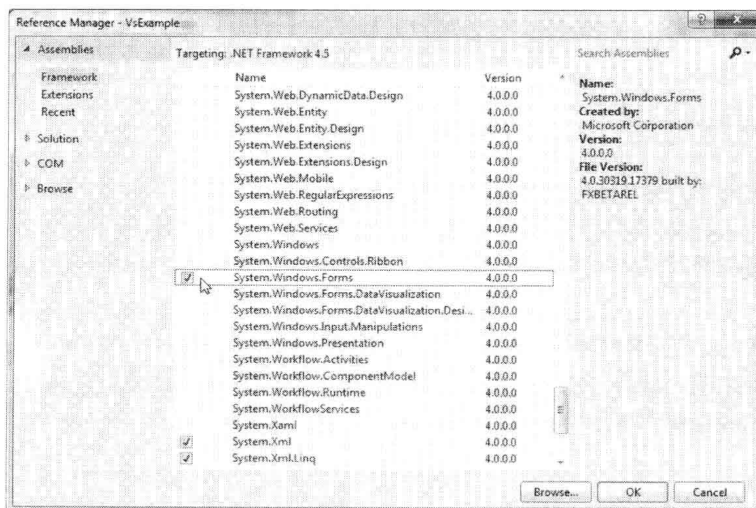


图2-10 Add Reference对话框

我们来简单测试一下。在Fremework项中找到System.Windows.Forms.dll程序集，选中相关的复选框，关闭对话框后会在Solution Explorer的References文件夹中看到该库。在Solution Explorer中选中该程序集，按下键盘上的Delete键（或鼠标右键选择Delete菜单项），可以删除该引用。

2. 查看Project Properties

最后，注意在解决方案资源管理器内有一个名为Properties的图标。当你双击该图标时，会呈现一个增强的项目配置编辑器（如图2-11所示）。

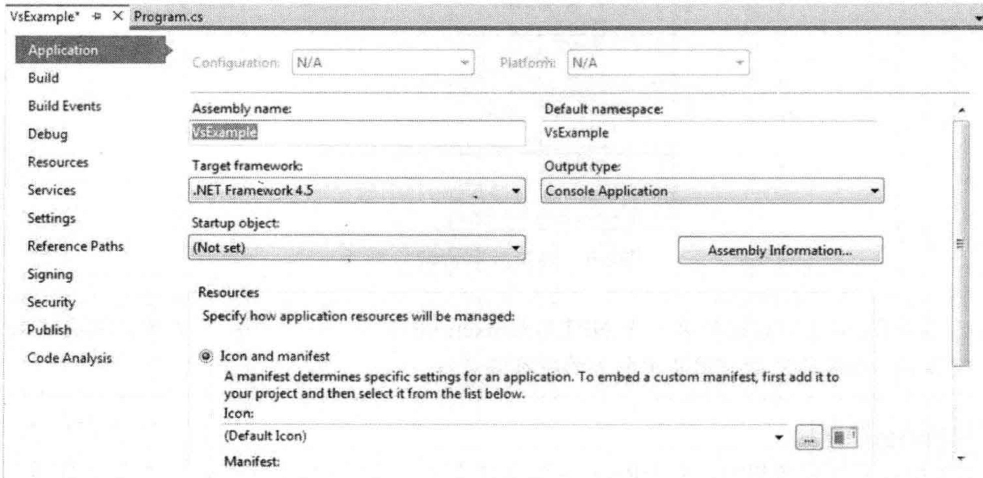


图2-11 Project Properties窗口

后面在其他内容的讲述中你将看见Project Properties窗口的各个方面。但是，只需花点时间研究一下，就会明白在这个窗口可以建立各种安全设置、强命名程序集（参见第14章）、部署应用程序、插入应用程序资源并对预先和事后构建的事件进行配置。

2.6.4 Class View工具

下一个要讨论的工具是Class View工具，可以在View菜单下使用该功能。该工具的目的是从面向对象角度（而不是从解决方案资源管理器的基于文件的角度）显示当前项目里的所有类型。顶部的窗格显示了一组命名空间及其类型的集合，而底部的窗格显示了当前所选择的类型的各个成员（如图2-12所示）。

当你在Class View工具中双击类型或类型成员时，Visual Studio将自动打开正确的C#代码文件，并将鼠标光标定位到正确的地方。Visual Studio的Class View工具还有一个非常好的特性，那就是你可以打开任何被引用的程序集，查看它包含的命名空间、类型和成员。

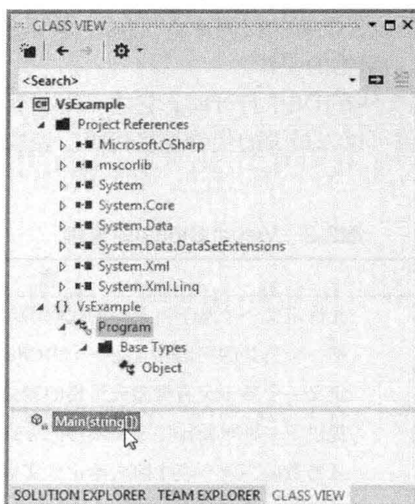


图2-12 Class View工具

2.6.5 Object Browser工具

Visual Studio也提供了一个工具，用于检查在当前项目内被引用的程序集。通过View菜单启动Object Browser，然后选择希望检查的程序集（如图2-13所示）。

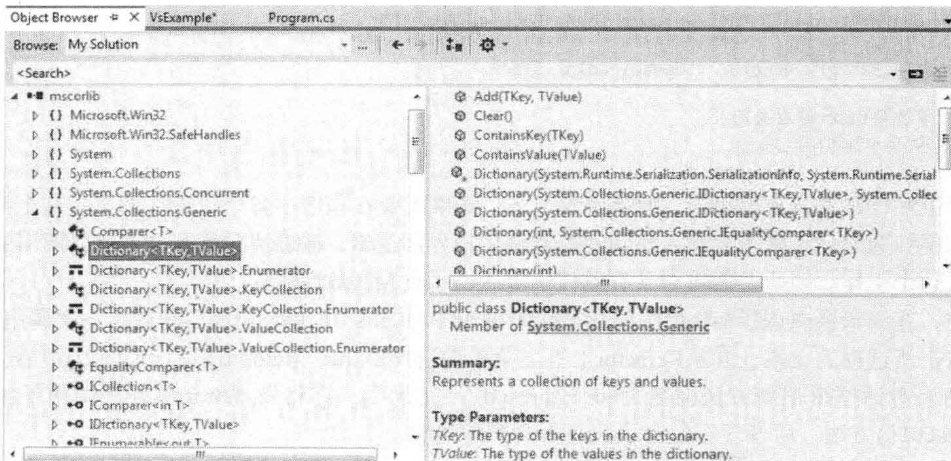


图2-13 Object Browser工具

2.6.6 集成对代码重构的支持

Visual Studio所配备的一个主要功能是，对既有代码进行重构的支持。简单地说，重构（refactoring）

是一个正规、严格的过程，目的是改善既有的代码库。[1]过去，代码的重构通常需要大量的手工劳动。幸好Visual Studio可以完成大量自动重构的工作。

在Refactor菜单（只有当代码文件在IDE中打开时，它才可用）下有相关的键盘快捷键、智能标签以及与上下文敏感的鼠标点击，你可以以最少的代价重组代码。表2-2定义了可以由Visual Studio识别的一些常见的重构。

表2-2　Visual Studio的重构

重构技术	作　用
Extract Method（提取方法）	允许定义一个基于所选择的代码语句的新方法
Encapsulate Field（封装字段）	把一个公共的字段转化为一个由C#属性封装的私有字段
Extract Interface（提取接口）	定义一个基于现有类型成员集的新接口类型
Reorder Parameters（重排参数）	提供了一种重新排序成员实参的方式
Remove Parameters（除去参数）	从参数的当前列表中除去给定的实参（如你期望的那样）
Rename（重命名）	允许在整个项目期间重新命名代码标记（方法名字、字段、本地变量等）

为了阐明重构是如何实现的，用以下代码修改Main()方法：

```
static void Main(string[] args)
{
  // 建立Console UI(CUI)
  Console.Title = "My Rocking App";
  Console.ForegroundColor = ConsoleColor.Yellow;
  Console.BackgroundColor = ConsoleColor.Blue;
  Console.WriteLine("*************************************");
  Console.WriteLine("***** Welcome to My Rocking App *****");
  Console.WriteLine("*************************************");
  Console.BackgroundColor = ConsoleColor.Black;

  // 等待按Enter键以关闭
  Console.ReadLine();
}
```

虽然上述代码本身正确无误，但是想象一下，如果需要在程序中各个位置上都显示这个提示，情形将会怎样？为避免不断重复输入相同的控制台用户界面逻辑，理想情况是有一个可以调用的辅助函数来完成这个工作。因此，你将对既有的代码应用Extract Method重构。

首先，在编辑器内选择Main()中的各个代码语句（对Console.ReadLine()的最后一次调用除外）。然后，右击所选择的文本，并从Refactor上下文菜单中选择Extract Method选项，如图2-14所示。

在结果对话框里把新方法命名为ConfigureCUI。完成后，你将发现Main()方法调用了新生成的ConfigureCUI()方法，后者现在包含了前面所选择的代码。

```
class Program
{
  static void Main(string[] args)
  {
    ConfigureCUI();
```

① 有关重构的进一步信息，请阅读Martin Fowler的《重构：改善既有代码的设计（英文注释版）》（人民邮电出版社，2010）。——编者注

```
      // 等待按键以关闭
      Console.ReadLine();
    }

    private static void ConfigureCUI()
    {
      // 建立Console UI(CUI)
      Console.Title = "My Rocking App";
      Console.ForegroundColor = ConsoleColor.Yellow;
      Console.BackgroundColor = ConsoleColor.Blue;
      Console.WriteLine("*************************************");
      Console.WriteLine("***** Welcome to My Rocking App *****");
      Console.WriteLine("*************************************");
      Console.BackgroundColor = ConsoleColor.Black;
    }
  }
```

这是使用Visual Studio内置的重构的一个简单示例，下面以及本书其他地方还有另外的示例。不过，你现在可以随意激活其他重构选项，看看它们各自有什么作用（不过本书后面不会再使用当前的VsExample项目了，所以也无需花费太多时间）。

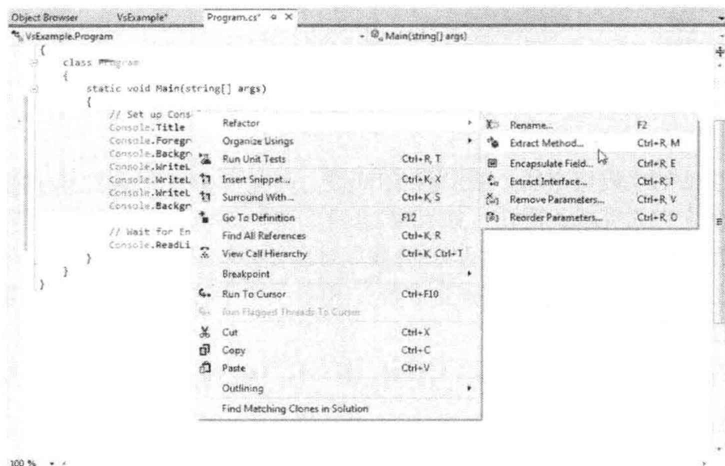


图2-14　激活代码重构

2.6.7　代码扩展和围绕技术

Visual Studio（以及Visual C# Express）也具有使用菜单选项、上下文关联的鼠标单击以及键盘快捷键插入预制的C#代码块的能力。可用的代码扩展数量是惊人的，并可分为两个大类。

- **片段**：这类模板在鼠标光标位置处插入公共代码块。
- **围绕**：这类模板在一个相关作用域内封装被选定语句的块。

亲身体验一下这个功能，假设你希望使用foreach语句来迭代Main()方法的传入参数。可以激活foreach代码块，而不用手动键入代码。激活之后，IDE会在鼠标光标的当前位置处弹出一个foreach代码模板。

举例说明，将鼠标光标放在Main()的第一个左花括号的后面。通过右击鼠标，选中Insert Snippet（或Surround With）菜单选项来激活代码块。将会在这里看到所有此类代码块的列表（按Esc键以消除弹出菜单），也可以只键入代码块的名字作为快捷方式，在本例中这个名字为foreach。如图2-15所示，注意代码块的图标看起来有点像一张撕碎的纸。

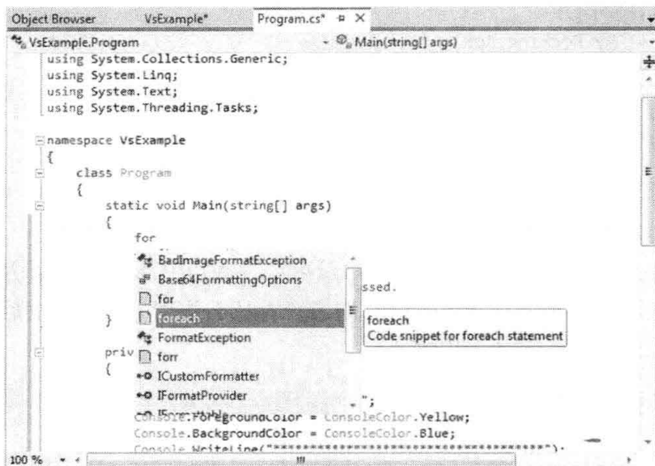


图2-15　激活代码块

找到了你希望激活的代码块之后，就按两次Tab键。它会自动完成整个代码块并且留下一组占位符，然后你就可以填充它来完成代码块。如果按Tab键，就可以切换每一个占位符并且填充内容（按Esc键来退出代码段编辑模式）。对于我们的实例，我们可以使用string数据类型改变第一个占位符（var关键字），使用arg改变第二个占位符（item变量名），用下一子字符串数组参数的名称改变最后一个占位符，得到的结果如下所示：

```
static void Main(string[] args)
{
  foreach (string arg in args)
  {

  }
...
}
```

如果如果在打开的C#代码文件中右击并选择Surround With菜单，同样地，你面前会呈现一连串的选项。还记得吗？在使用围绕代码块的时候，我们通常先选择一段代码语句块来表示要被包装的东西（如try/catch块等）。要确保花一点儿时间浏览这些预定义的代码扩展模板，因为它们能够大幅度加快开发的速度。

说明　所有代码扩展模板都是对IDE内代码基于XML的描述。使用Visual Studio（以及Visual C# Express）可以创建自定义的代码模板。具体实现细节可以到http://msdn.microsoft.com/en-US/library/ms379562上查看我的文章“Investigating Code Snippet Technology”。

2.6.8 可视化Class Designer

Visual Studio具备可视化设计类和其他类型的能力（但Visual C# Express不具备）。这种Class Designer工具允许在项目里查看和修改各个类型的相互关系（类、接口、结构、枚举和委托）。使用这种工具，能够可视地对一个类型增加（或者去除）成员，并且所做的修改能反映在对应的C#源代码文件里。同样，当修改一个给定的C#源代码文件时，所做的修改会反映在对应的类图里。[①]

为了能够使用Visual Studio的这一功能，第一步是插入一个新的类图文件。这有许多方法可以实现，其中一个方法就是单击位于Solution Explorer右侧的View Class Diagram按钮（如图2-16所示，确保所选中的是项目，而不是解决方案）。

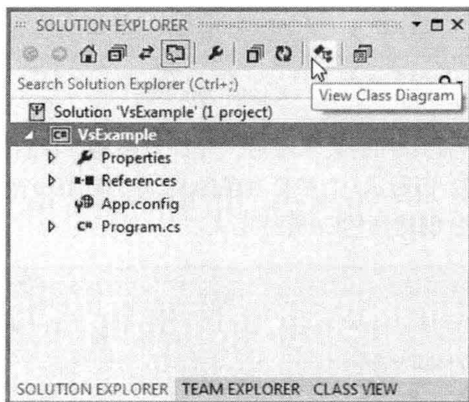


图2-16 向当前项目插入类图文件

完成之后，在当前项目里将出现代表各个类型的类图标。单击箭头图标可以显示或隐藏类型成员（如图2-17所示）。

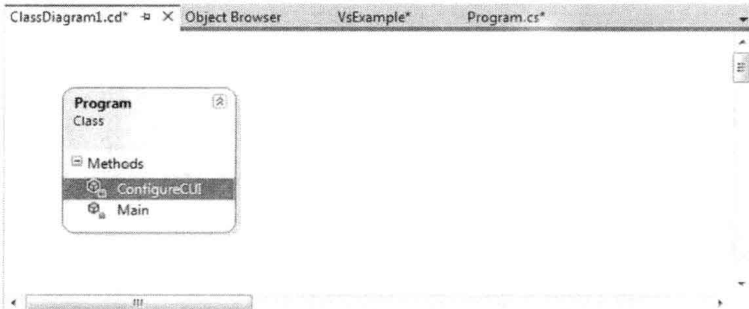


图2-17 Class Diagram查看器

该工具与Visual Studio的其他两个功能结合发挥作用：Class Details窗口（通过View→Other Windows菜单启动）以及Class Designer Toolbox（通过View→Toolbox菜单项启动）。Class Details窗口

① 这就是所谓的模型驱动开发（Model-Driven Development，MDD）的基本实现。——编者注

不仅显示图中当前所选项的细节，而且还允许在执行状态下修改已有成员并插入新成员（如图2-18所示）。

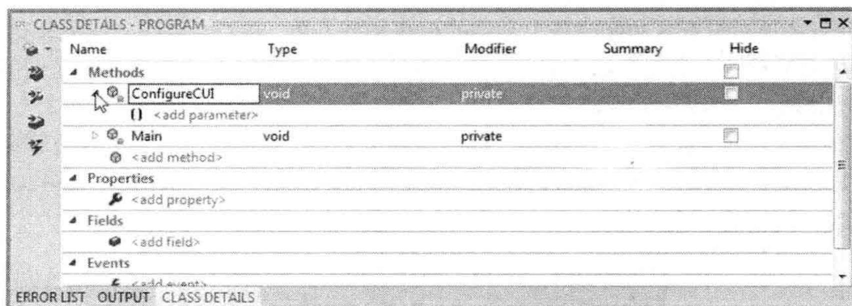


图2-18　Class Details窗口

Class Designer Toolbox也能使用View菜单激活，允许可视地在项目里插入新的类型（并且在这些类型之间创建关联关系），如图2-19所示。（需要注意的是，必须有一个类图作为活动窗口来查看该工具箱。）完成后，IDE自动在后台创建新的C#类型定义。

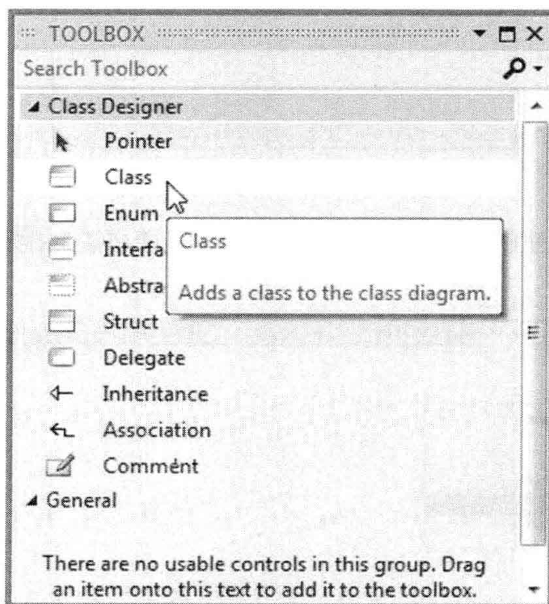


图2-19　Class Designer Toolbox

举个例子，从Class Designer Toolbox里拖一个新的Class到Class Designer上。在结果对话框里将该类命名为`Car`。接着，使用Class Details窗口，增加一个名为`petName`的公共`string`字段（如图2-20所示）。

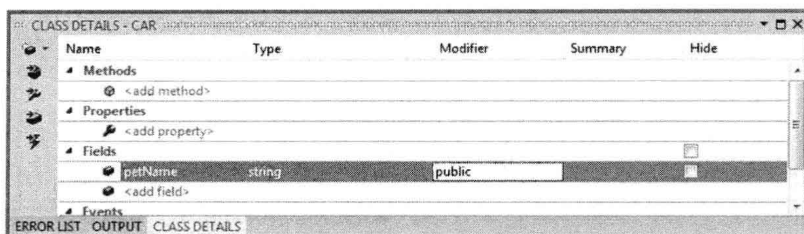


图2-20 用Class Details窗口增加字段

如果现在查看Car类的C#定义，你会看到它已经相应地被修改了（除代码注释以外）:

```
public class Car
{
  // 一般不应使用公共数据，这里之所以破例，是为了使代码简单一些
  public string petName;
}
```

再次激活设计器文件，在设计器中增加另一个新的类SportsCar。现在，从Class Designer Toolbox里选择Inheritance图标并单击SportsCar图标的顶部。按住鼠标左键不放，把鼠标移到Car类图标的顶部后松开。一切无误的话，就已经从Car派生出了SportsCar类（如图2-21所示）。

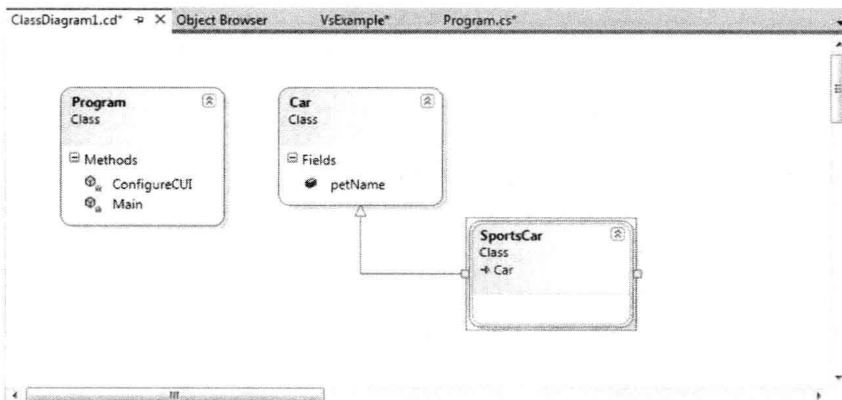


图2-21 从现有的类中可视地派生新的类

为了完成本例，用一个名为GetPetName()的公共方法更新所生成的SportsCar类，如下所示：

```
public class SportsCar : Car
{
  public string GetPetName()
  {
    petName = "Fred";
    return petName;
  }
}
```

在阅读本书的过程中，你可以使用Visual Studio中的这些可视化工具。现在，你应该已经对IDE的基础知识有了充分的了解。

说明　我们将在第6章全面研究关于继承的概念。

2.6.9 集成的.NET Framework 4.5 SDK文档系统

Visual Studio的最后一个功能肯定会让你从一开始就感觉很舒服，那就是完全集成的帮助系统。.NET Framework 4.5 SDK的文档系统非常出色，可读性很强，并且包含充足的有用信息。由于有大量预定义的.NET类型（数量有上千个之多），你必须准备深入学习文档系统。如果不这样做，作为.NET开发人员，你注定会承受一个漫长、令人沮丧而痛苦的过程。

如果你能够上网，就可以在网址http://msdn.microsoft.com/library在线查看.NET Framework 4.5 SDK文档。

在主页中，使用树形视图导航找到.NET Development，点击后选择.NET Framework 4.5，之后你就可以看到所有的.NET Framework Class Library的重要链接了，每个链接都用单独的.NET命名空间记录了该类型（如图2-22所示）。

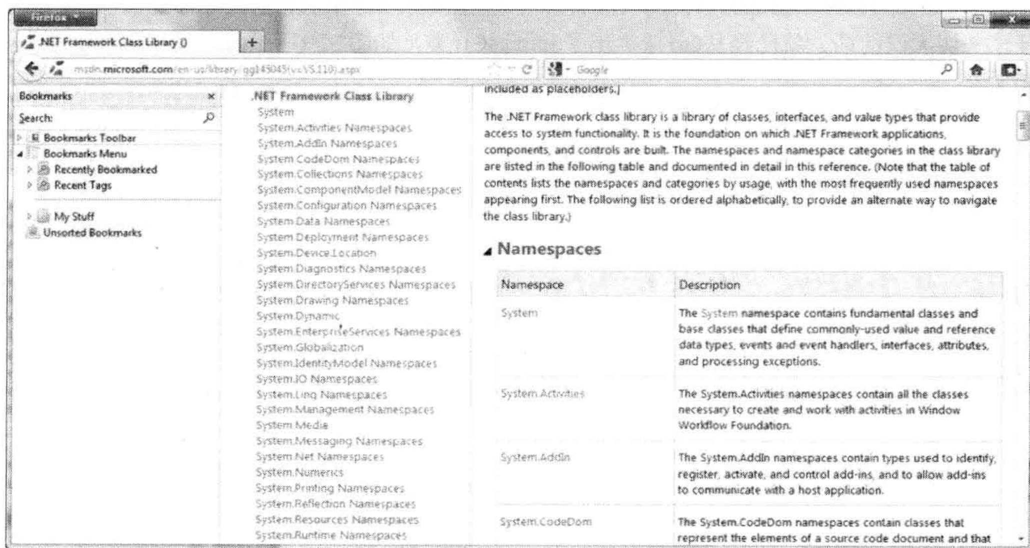


图2-22　在线查看.NET Framework 4.5文档

在安装Visual Studio的过程中，你可以在计算机上安装同样的帮助系统（如果你不能上网，这一设置就非常实用）。要本地安装帮助系统，单击Windows的“开始”按钮，选择“All Programs”→“Microsoft Visual Studio 11”→“Microsoft Help Viewer”工具，然后选择添加你感兴趣的帮助文档，如图2-23所示（如果硬盘空间允许，我建议你添加所有的文档）。

不论是本地系统还是在线，与文档交互的最简单方式是选择一个C#关键字，在Visual Studio代码窗口中输入名称或成员名称，按下F1键。这将自动打开所选项的文档窗口。例如，在Car类的定义中选择string关键字。按下F1键时，将出现Help页面。

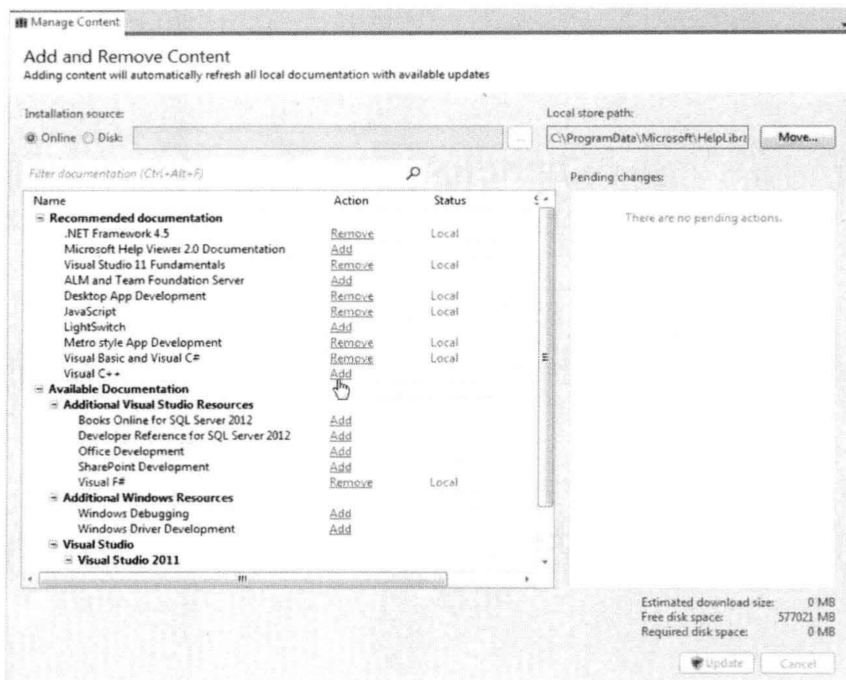


图2-23 Help Library Manager允许你下载.NET Framework 4.5 SDK文档

帮助文档另一个有用的地方是位于左上方的Search编辑框。你可以输入任何命名空间、类型或成员的名称，它将为你导航到正确的位置。如果你要检索System.Reflection命名空间，就可以学习该命名空间的详细内容，研究它包含的类型，查看代码示例等。

说明 听上去可能有点像复读机，但学习使用.NET Framework 4.5 SDK文档真的非常重要。不管是什么书，不管它有多厚，都无法涵盖.NET平台的全部内容。所以，花点时间充分了解帮助系统吧，以后你会庆幸这个决定的。

2.7 小结

可以看到，有很多新工具！本章概述了C#程序员在开发期间可能用到的各种主要编程工具。我们首先学习了如何仅使用免费的C#编译器（cxc.exe）和记事本（Notepad/ Notepad++）生成.NET程序集。接着展示了如何使用这些工具编辑和编译*.cs代码文件的过程。

本章还讨论了3个功能丰富的IDE，开始是开源的SharpDevelop，接着是微软的Visual C# Express和Visual Studio Professional。本章仅初步讨论了各个工具的功能，但有益于你进一步探讨你所选定的IDE。在学习本书的过程中，你还会发现Visual Studio的其他功能。

Part 2

第二部分

C# 核心编程结构

本部分内容

第3章

C#核心编程结构 I

本章通过介绍很多小而独立的主题来开始对C#编程语言的正式研究，这些主题是我们在探索.NET框架时必须了解的。第一件事情就是理解如何构建程序的*应用程序对象*以及可执行程序入口点Main()方法的构成。接着，我们会研究C#基本数据类型（和它们在System命名空间中的等价类型），包括System.String和System.Text.StringBuilder类类型。

了解了基本.NET数据类型的细节之后，我们会研究一些数据类型转换的技术，包括窄化运算、宽化运算以及checked和unchecked关键字的使用。

本章还将介绍C#关键字var的作用，它允许你*隐式地*定义本地变量。你将看到，在使用LINQ技术时，隐式类型是非常有帮助的。最后我们会快速过一遍允许你使用循环和选择结构控制应用程序流程的C#关键字和操作符。

3.1 一个简单的C#程序

C#要求所有的程序逻辑都包含在一个类型定义中（第1章中我们讲过，*类型*是指集合{类，接口，结构，枚举，委托}中的一个成员）。与其他语言不同，要在C#中创建全局函数或全局数据是不可能的。但所有的成员和方法都必须包含在一个类型定义中。首先，新建一个控制台应用程序项目SimpleCSharpApp。初始的Program.cs文件中的代码很简单，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimpleCSharpApp
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

用以下的语句修改Main()方法：

```
class Program
{
```

```
    static void Main(string[] args)
    {
      // 显示一条简单的消息
      Console.WriteLine("***** My First C# App *****");
      Console.WriteLine("Hello World!");
      Console.WriteLine();

      // 按下Enter键以后关闭
      Console.ReadLine();
    }
  }
```

3

说明 C#是一种区分字母大小写的程序设计语言。所以，Main与main不同，Readline与ReadLine不同。因此，要注意所有C#关键字都是小写的（如public、lock、class和dynamic等），同时命名空间、类型和成员名称（按约定）以一个大写字母开头，中间的单词都是首字母大写（如Console.WirteLine、System.Windows.MessageBox、System.Data.SqlClient等）。按照惯例，如果出现一个"undefined symbols"编译错误提示，就需要检查一下拼写是否正确。

在这里，我们定义了只有一个Main()方法的类类型。默认情况下，Visual Studio会把定义Main()的类命名为"Program"，然而，如果需要的话我们完全可以修改名字。每一个可执行的C#应用程序（控制台程序、Windows桌面程序或Windows服务）必须包含一个定义了Main()方法的类，这个方法用来表示应用程序的入口点。

正式地说，定义Main()方法的类叫做*应用程序对象*。虽然一个可执行程序可以有多个应用程序对象（在执行单元测试的时候可能有用），但是我们必须通过命令行编译器的/main选项或通过位于Visual Studio项目属性编辑器中Application选项卡内的Startup Object下拉列表框（参见第2章）来通知编译器将哪个Main()方法作为入口点。

注意Main()的签名[①]具有static关键字，我们会在第5章中研究其细节。在这里，只需要理解静态成员是类级别的（而不是对象级别的），因此在调用之前不需要先创建新的类实例。

除了static关键字之外，这个Main()方法还有一个参数，是一个字符串数组（string[] args）。现在我们还不必为处理这个数组操心，这个参数可以包含任意数量的命令行输入参数（你马上就会看到怎样访问它们）。最后，Main()方法使用了void返回值，也就是说我们不需要在退出方法作用域之前使用return关键字来显式定义一个返回值。

Program的逻辑在Main()里面。在这里使用了在System命名空间中定义的Console类。你可能会猜到，在它的成员集合中有一个静态的WriteLine()，能够将一个文本字符串和回车符输送到标准输出端。还可以调用Console.ReadLine()，保证由Visual Studio IDE启动的命令提示符在调试的会话期间保持可见[②]，直到按下Enter键。我们稍后就会详细介绍System.Console类。

3.1.1 Main()方法的其他形式

默认状态下，Visual Studio生成的Main()有一个void返回值，并且只接受一个参数（一个字符串数

① 所谓签名，就是指一个方法的名称、返回类型和参数列表。——编者注

② 即不会立即关闭控制台窗口，让你可以查看输出结果。——译者注

组）。但这不是唯一可能的Main()形式。使用下列任何一个签名（假设它包含在一个C#类或结构定义中）构造应用程序的入口点都是允许的：

```
// 整型返回类型，以字符串数组作为参数
static int Main(string[] args)
{
  // 在退出之前必须返回一个值
  return 0;
}

// 没有返回类型，没有参数
static void Main()
{
}

// 整型返回类型，没有参数
static int Main()
{
  // 在退出之前必须返回一个值
  return 0;
}
```

说明　如果我们不提供一个明确的访问修饰符，Main()默认就是私有的，但它也可以被定义为公有的。Visual Studio会把程序的Main()方法自动定义为隐式私有的，以确保其他应用程序不能直接调用另一个应用程序的入口点。

很明显，选择怎样构造Main()要基于两个问题。第一，当Main()完成并且程序终止时，是否要向系统返回一个值？如果是，需要返回一个int数据类型而不是void。第二，是否需要处理用户提供的命令行参数？如果是，它们将被保存在string数组中。下面我们来深入了解一下不同的情况。

3.1.2　指定应用程序错误代码

虽然绝大多数Main()方法会以void作为返回值，但是C#和其他C系列的语言一样，都可以从Main()返回一个int。根据惯例，返回值0表示程序正常结束，而其他值（如–1）则表示有错误发生（要知道，值0是自动返回的，即使Main()方法的原型结构返回void）。

在Windows操作系统上，应用程序的返回值保存在一个叫做%ERRORLEVEL%的系统环境变量中。如果我们要创建一个以编程方式启动另外一个可执行程序的应用程序（第17章会研究这个主题），就可以使用静态的System.Diagnostics.Process.ExitCode属性来获取%ERRORLEVEL%的值。

由于应用程序的返回值是在程序结束时传给系统，显然应用程序不太可能在程序运行时获取并显示最终的错误代码。然而，为了演示如何查看程序终止时的错误级别，我们先按如下所示来更新Main()方法：

```
// 注意，现在返回int，而不是void
static int Main(string[] args)
{
  // 显示一个消息并等待按Enter键
  Console.WriteLine("***** My First C# App *****");
  Console.WriteLine("Hello World!");
```

```
    Console.WriteLine();
    Console.ReadLine();

    // 任意返回一个错误代码
    return -1;
  }
```

现在借助批处理文件来捕获Main()的返回值。使用Windows资源管理器，转到包含已编译的应用程序的目录（如C:\SimpleCSharpApp\bin\Debug）。在Debug文件夹中新建一个文本文件（命名为SimpleCSharpApp.bat）并包含如下的代码（如果之前没有编写过*.bat文件，不要过分关心其中的细节，这只是一个测试）：


```
@echo off

rem A batch file for SimpleCSharpApp.exe
rem which captures the app's return value.

SimpleCSharpApp
@if "%ERRORLEVEL%" == "0" goto success

:fail
  echo This application has failed!
  echo return value = %ERRORLEVEL%
  goto end
:success
  echo This application has succeeded!
  echo return value = %ERRORLEVEL%
  goto end
:end
  echo All Done.
```

至此，打开Visuat Studio命令提示符，转到包含可执行文件以及新的*.bat文件的目录。通过输入批处理文件名字并按Enter键来执行批处理。假设Main()方法返回-1，应该能看到如下所示的输出。如果Main()方法返回0，可能会见到“This application has succeeded!”这样的消息被输出到控制台。

```
***** My First C# App *****
Hello World!

This application has failed!
return value = -1
All Done.
```

再说一次，绝大多数（但不是全部）C#应用程序会使用void作为Main()的返回值，你应该记得吧，这其实是在隐式返回0作为错误代码。为此，本书中使用的Main()方法都会返回void。（其余的项目当然不需要使用批处理文件来捕获返回值！）

3.1.3 处理命令行参数

既然你对Main()方法的返回值有了更好的了解，那么就来研究字符串数据的传入数组。假设希望更新应用程序来处理可能的命令行参数，其中一个方法是使用C#的for循环（本章后面会研究C#的迭代结构）：

```
static int Main(string[] args)
{
...
  // 处理传入的参数
  for(int i = 0; i < args.Length; i++)
    Console.WriteLine("Arg: {0}", args[i]);

  Console.ReadLine();
  return -1;
}
```

在这里，使用System.Array的Length属性，检查string数组是否包含一些项。在第4章中你将会看到，所有C#数组实际上都是System.Array类的别名，因此具有一组公共的成员集合。当循环到数组的每一项时，它的值被输出到控制台窗口中。在命令行提供参数也一样简单，如下所示。

```
C:\SimpleCSharpApp\bin\Debug>SimpleCSharpApp.exe /arg1 -arg2

***** My First C# App *****
Hello World!
Arg: /arg1
Arg: -arg2
```

作为标准的for循环的替代方案，可以用C#的foreach关键字迭代传入的string数组。这里是一些示例用法：

```
// 注意，当使用foreach时不需要检查数组的大小
static int Main(string[] args)
{
...
  // 使用foreach处理所有传入的参数
  foreach(string arg in args)
    Console.WriteLine("Arg: {0}", arg);

  Console.ReadLine();
  return -1;
}
```

最后，还可以使用System.Environment类型的静态方法GetCommandLineArgs()访问命令行参数。这个方法的返回值是一个string数组。第一个索引表示应用程序本身的名称，而数组中其余的元素包含单独的命令行参数。当使用这个方法时，不再需要将Main()定义成输入参数为一个string数组的方法，虽然这样做没有害处。

```
static int Main(string[] args)
{
...
  // 用System.Environment获取参数
  string[] theArgs = Environment.GetCommandLineArgs();
  foreach(string arg in theArgs)
    Console.WriteLine("Arg: {0}", arg);

  Console.ReadLine();
  return -1;
}
```

当然，程序会对哪些命令行参数（如果有的话）做出响应以及参数是如何格式化的（例如加“-”

或“/”前缀），这都取决于你。在这里，我们只是传入一系列选项并直接输出到命令提示符。然而，假设我们在创建一个视频电子游戏并且编写程序来处理一个叫做-godmode的选项。如果用户使用这个标志启动应用程序，我们可以知道这个用户其实是一个作弊者，并可以采取适当措施。

3.1.4　使用Visual Studio指定命令行参数

在现实世界中，启动程序时，要由用户提供该程序使用的命令行参数。但是，在开发周期中，程序员可能出于测试目的希望指定一些命令行标志。在Visual Studio要想这样做，就应在Solution Explorer中双击Properties图标，选择左侧的Debug（调试）标签。在这里，使用Command line arguments（命令行参数）文本框指定值（如图3-1所示）。

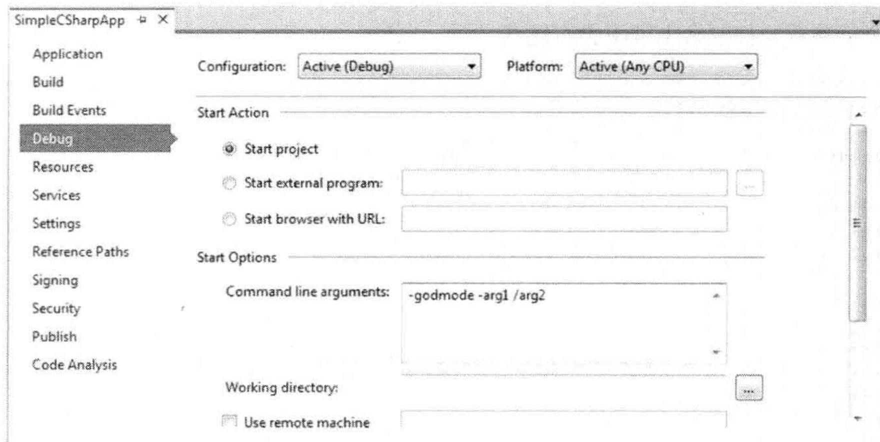


图3-1　通过Visual Studio设置命令行参数

一旦建立了这样的命令行参数，在Visual Studio IDE中调试或运行应用程序的时候，它们将被自动传递给Main()方法。

3.2　有趣的题外话：System.Environment 类的其他成员

除了GetCommandLineArgs()以外，Environment类型还有一些有用的方法。这个类允许我们通过不同的静态成员获得大量有关运行.NET应用程序的操作系统的细节。为了说明这个类的有效性，修改Main()方法以调用名为ShowEnvironmentDetails()的辅助方法：

```
static int Main(string[] args)
{
...
  // Program类的辅助方法
  ShowEnvironmentDetails();

  Console.ReadLine();
  return -1;
}
```

在Program类中实现这个方法来调用Environment类型的各种成员。例如：

```
static void ShowEnvironmentDetails()
{
  // 输出本机的驱动器以及其他一些有用的细节信息
  foreach (string drive in Environment.GetLogicalDrives())
    Console.WriteLine("Drive: {0}", drive);

  Console.WriteLine("OS: {0}", Environment.OSVersion);
  Console.WriteLine("Number of processors: {0}",
    Environment.ProcessorCount);
  Console.WriteLine(".NET Version: {0}",
    Environment.Version);
}
```

下面的输出显示了调用这个方法的一个可能的测试运行。如果你没有通过Visual Studio的Debug标签来指定命令行参数，控制台上就不会有这些。

```
 ***** My First C# App *****

Hello World!

Arg: -godmode
Arg: -arg1
Arg: /arg2

Drive: C:\
Drive: D:\
Drive: E:\
Drive: F:\
Drive: G:\
Drive: H:\
Drive: I:\
OS: Microsoft Windows NT 6.1.7601 Service Pack 1
Number of processors: 4
.NET Version: 4.0.30319.17020
```

除了前一个示例中出现过的成员，Environment类型还定义了其他成员。表3-1给出了另外一些有趣的属性。但是，查阅.NET Framework 4.5 SDK文档才能了解全部的细节。

表3-1 System.Environment的部分属性

属　性	作　用
ExitCode	获取或设置应用程序中任何地方的退出代码
Is64BitOperatingSystem	返回布尔值，代表主机是否运行64位操作系统
MachineName	获得当前机器的名字
NewLine	获得当前环境的换行符
SystemDirectory	返回通向系统目录的完整路径
UserName	返回启动这个应用程序的用户的名称
Version	返回版本对象，代表.NET平台的版本

源代码 SimpleCSharpApp项目的源代码位于Chapter 3子目录下。

3.3 System.Console 类

前面几章中所创建的示例应用程序都广泛地使用了System.Console类。虽然CUI(命令行用户界面)不如GUI或基于Web的前端那样吸引人，但是将早期的示例限制为只使用CUI，我们就可以将重点集中在C#语法和.NET平台的核心方面，而不用处理建立GUI或网站的复杂性。

顾名思义，Console类封装了基于控制台应用程序的输入、输出和错误流操作。表3-2列出了一些(但不是全部)比较受关注的成员。从表中可以看出，Console类的许多成员能为简单的命令行程序“增添情趣”，例如，可以改变背景颜色和前景颜色，以各种频率发出蜂鸣声。

表3-2 System.Console的部分成员

成　员	作　用
Beep()	这个方法强制控制台发出指定频率和持续时间的蜂鸣声
BackgroundColor ForegroundColor	这些属性设置当前输出的背景/前景色。它们可以被赋予ConsoleColor枚举的任何成员
BufferHeight BufferWidth	这些属性控制控制台缓冲区域的高度/宽度
Title	这个属性设置当前控制台的标题
WindowHeight WindowWidth WindowTop WindowLeft	这些属性控制与已建立的缓冲区相关的控制台大小
Clear()	这个方法清除已建立的缓冲区和控制台的显示区域

3.3.1 使用Console类进行基本的输入和输出

除了表3-2的成员以外，Console类还定义了捕获输入和输出的一套方法，它们都被定义成静态的，因此能够通过将类的名字作为方法名的前缀来调用。可以看到，WriteLine()将文本字符串(包括一个回车符)输送到输出流。Write()方法将文本输送到输出流而不附加回车符。ReadLine()从输入流接收信息直到遇到回车符，而Read()被用来从输入流获得一个字符。

为了演示使用Console类进行基本的I/O，我们来新建一个控制台应用程序BasicConsoleIO，并且更新Main()方法来调用GetUserData()辅助方法：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Basic Console I/O *****");
    GetUserData();
    Console.ReadLine();
  }
}
```

说明　在第2章中，我们简要介绍了Visual Studio代码片段。在本书前面几章中，cw代码片段十分有用，因为它能自动扩展为Console.WriteLine()！想自己测试一下，你可以在Main()方法中输入cw，再按两次Tab键。可惜的是，没有Console.ReadLine()的代码片段。

在Program类中实现这个方法，采用的逻辑是，提示用户一些信息并且通过标准的输出流来回显每一项。例如，我们可以询问用户的姓名和年龄（为了简单起见，我们把它当做文本值，而不是我们期望的数值），代码如下：

```
static void GetUserData()
{
  // 获取姓名和年龄
  Console.Write("Please enter your name: ");
  string userName = Console.ReadLine();
  Console.Write("Please enter your age: ");
  string userAge = Console.ReadLine();

  // 改变回复的颜色，只是为了好玩
  ConsoleColor prevColor = Console.ForegroundColor;
  Console.ForegroundColor = ConsoleColor.Yellow;

  // 回复到控制台
  Console.WriteLine("Hello {0}!  You are {1} years old.",
    userName, userAge);

  // 恢复之前的颜色
  Console.ForegroundColor = prevColor;
}
```

不出所料，运行程序后，输入数据就被输出到了控制台（使用自定义的颜色）。

3.3.2　格式化控制台输出

在前面的这几章中，多次见到诸如{0}、{1}之类的标记嵌入在字符串字面量中。.NET引入了一种字符串格式化的新风格，与C语言的printf()语句相似。简而言之，如果需要定义一个字符串字面量，其中包含一些要到运行时才能知道其值的数据片段，可以使用这种花括号语法在文本内部指定占位符。在运行时，值会传入到Console.WriteLine()来替代每一个占位符。

传给WriteLine()的第一个参数代表一个包含由{0}、{1}、{2}等（大括号占位符数字编码方式总是以0开始）指定的可选占位符的字符串字面量。其余传给WriteLine()的参数就是要插入各自占位符的值。

说明　如果唯一编号的大括号占位符的数量比填充的参数数量多，则会在运行时收到一个格式异常；若是比填充的参数数量少，没有使用的填充参数就会被忽略。

在一个给定的字符串中可以重复特定的占位符。举例来说，如果你是一个甲壳虫乐队的爱好者，想建立一个字符串"9, Number 9, Number 9"[①]，你会写：

① 这是甲壳虫乐队*Revolution No. 9*这首歌曲的歌词，下面注释中的John是该乐队的已故主唱约翰・列侬。
——编者注

```
// John说……
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

还要知道，可以在字符串字面量的任意位置放置占位符，而不需要按照递增的次序。例如，看看如下的代码片段：

```
// 输出：20，10，30
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
```


3.3.3 格式化数值数据

如果数值数据需要更精细的格式化，每一个占位符都可以（可选地）包含不同的格式字符，表3-3显示了最常用的格式化选项。

表3-3 .NET数值格式字符

字符串格式字符	作 用
C或c	用于格式化货币。默认情况下，这个标志会以当地的货币符号为前缀（美国英语中是一个美元符号[$]）
D或d	用于格式化十进制数。这个标志还可以用于指定填充值的最小个数
E或e	用于指数记数法。无论指数常数是大写（E）还是小写（e），都进行转换控制
F或f	用于定点小数的格式化。这个标志也用于指定填充值的最小个数
G或g	代表general。这个字符能用来将一个数格式化为定点或指数格式
N或n	用于基本的数值格式化（带逗号）
X或x	用于十六进制格式化。如果使用大写的X，十六进制格式也会包含大写的字符

给定的占位符值以冒号为标记，将这些字符作为后缀（例如，{0:C}、{1:d}、{2:X}）。修改Main()方法以调用一个名为FormatNumericalData()的新辅助函数，从而格式化一个固定值。实现方法有很多，例如：

```
// 使用一些格式化标记
static void FormatNumericalData()
{
  Console.WriteLine("The value 99999 in various formats:");
  Console.WriteLine("c format: {0:c}", 99999);
  Console.WriteLine("d9 format: {0:d9}", 99999);
  Console.WriteLine("f3 format: {0:f3}", 99999);
  Console.WriteLine("n format: {0:n}", 99999);

  // 注意，十六进制数的大小写形式决定了字母是大写还是小写
  Console.WriteLine("E format: {0:E}", 99999);
  Console.WriteLine("e format: {0:e}", 99999);
  Console.WriteLine("X format: {0:X}", 99999);
  Console.WriteLine("x format: {0:x}", 99999);
}
```

下面的输出显示了调用FormatNumericalData()方法的结果。

```
The value 99999 in various formats:

c format: $99,999.00
d9 format: 000099999
f3 format: 99999.000
n format: 99,999.00
E format: 9.999900E+004
e format: 9.999900e+004
X format: 1869F
x format: 1869f
```

在本书中其他地方还会有其他格式化的示例。但如果你想进一步深入学习.NET字符串格式化，可以查阅.NET Framework 4.5 SDK文档中的Formatting Types（格式化类型）主题。

源代码 BasicConsoleIO项目的源代码位于Chapter 3子目录下。

3.3.4 在控制台应用程序外格式化数值数据

最后说明一下，.NET字符串格式化字符不局限于在控制台程序中使用！同样的格式化语法可以在调用静态的string.Format()方法时使用。如果我们需要对任何应用程序类型（如桌面GUI程序、ASP.NET Web程序）在运行时组合文本数据，这个方法就很有用。

string.Format()返回一个新的字符串对象，改对象根据提供的标志进行格式化。此后，我们就可以随意使用需要的文本数据了。例如，假设我们在构建一个图形化的WPF桌面应用程序，需要格式化一个字符串，用于在消息对话框中显示，下面的代码显示了如何做到，但需要注意，在引用PresentationFramework.dll程序集之后才能编译代码，这样才能在随后的项目中使用（参见第2章有关使用Visual Studio引用库的信息）：

```
static void DisplayMessage()
{
  //使用string.Format()来格式化字符串字面量
  string userMessage = string.Format("100000 in hex is {0:x}",
    100000);

  //可能需要引用PresentationFramework.dll才能编译这行代码
  System.Windows.MessageBox.Show(userMessage);
}
```

3.4 系统数据类型和相应的 C#关键字

和任何编程语言一样，C#定义了一组用于表示局部变量、成员变量、返回值以及输入参数的基本数据类型。然而，和其他编程语言不同的是，这些关键字不只是简单的编译器可以识别的标记。C#数据类型关键字其实是System命名空间中完整类型的简化符号。表3-4列出了每一个系统数据类型、它们的范围、对应的C#关键字以及类型是否遵循CLS（公共语言规范）。

说明　第1章介绍过，符合CLS的.NET代码可以被任何托管编程语言使用。如果在你的程序中出现了不符合CLS的数据，那么其他语言可能就不能使用它们。

表3-4　C#内建系统类型

C#简化符号	判断符合CLS	系统类型	范　围	作　用
bool	是	System.Boolean	true或false	表示真实的或者虚假的
sbyte	否	System.SByte	–128~127	带符号的8位数
byte	是	System.Byte	0~255	无符号的8位数
short	是	System.Int16	–32 768~32 767	带符号的16位数
ushort	否	System.UInt16	0~65 535	无符号的16位数
int	是	System.Int32	–2 147 483 648~2 147 483 647	带符号的32位数
uint	否	System.UInt32	0~4 294 967 295	无符号的32位数
long	是	System.Int64	–9 223 372 036 854 775 808~ 9 223 372 036 854 775 807	带符号的64位数
ulong	否	System.UInt64	0~18 446 744 073 709 551 615	无符号的64位数
char	是	System.Char	U+0000~U+ffff	一个16位的Unicode字符
float	是	System.Single	$-3.4 \times 10^{38} \sim +3.4 \times 10^{38}$	32位浮点数
double	是	System.Double	$\sim 5.0 \times 10^{-324} \sim \pm 1.7 \times 10^{308}$	64位浮点数
decimal	是	System.Decimal	$(-7.9 \times 10^{28} \sim 7.9 \times 10^{28})/(10^{0\sim 28})$	128位带符号数
string	是	System.String	受系统内存的限制	表示一个Unicode字符集合
object	是	System.Object	任何类型都可以保存在一个object变量中	.NET世界中所有类型的基类

说明　默认情况下，浮点数被当做double类型。为了声明一个float变量，使用后缀f或F可以在原始的数值后面加上f或F（如5.3F）；使用后缀m或者M可以将一个浮点数声明为十进制（如330.5M）。同样，原始整数在默认情况下为int数据类型。要将其设置为long类型，可使用后缀l或L（如4L）。

3.4.1　变量声明和初始化

如果需要声明一个数据类型作为局部变量（如成员体内的变量），可以通过在变量名之前指定数据类型来实现。首先创建一个名为BasicDataTypes的控制台应用程序项目。更新Program类，使其具有如下的辅助方法，此方法在Main()中调用：

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // 局部变量这样进行声明: 数据类型 变量名
  int myInt;
  string myString;
```

```
    Console.WriteLine();
}
```

要知道，如果在分配初始值之前使用局部变量，会收到一个编译器错误。因此，声明时为局部数据赋一个初始值是一个好的做法。我们可以在单行中这么做，也可以将声明和赋值分开到两行代码语句中：

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // 局部变量在如下代码中声明和初始化
  // 数据类型 变量名=初始值
  int myInt = 0;

  // 我们还可以在两行中声明和赋值
  string myString;
  myString = "This is my character data";

  Console.WriteLine();
}
```

还允许在一行代码中声明多个具有相同实际类型的变量，如下所示的3个布尔型变量：

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  int myInt = 0;
  string myString;
  myString = "This is my character data";

  // 在一行中声明3个布尔型数据
  bool b1 = true, b2 = false, b3 = b1;
  Console.WriteLine();
}
```

同样，由于C# bool关键字只是System.Boolean结构的简化符号，因此可以使用它们的全名来分配任何数据类型（当然，对于任何C#数据类型关键字来说，这点都是正确的）。以下是LocalVarDeclarations()的最终实现：

```
static void LocalVarDeclarations()
{
  Console.WriteLine("=> Data Declarations:");
  // 局部变量声明和初始化方式: 数据类型 变量名=初始值
  int myInt = 0;

  string myString;
  myString = "This is my character data";

  // 在一行声明3个布尔型数据
  bool b1 = true, b2 = false, b3 = b1;

  // 使用系统数据类型声明布尔型变量
  System.Boolean b4 = false;

  Console.WriteLine("Your data: {0}, {1}, {2}, {3}, {4}, {5}",
      myInt, myString, b1, b2, b3, b4);

  Console.WriteLine();
}
```

3.4.2　内建数据类型与new操作符

所有的内建数据类型都支持默认构造函数（见第5章）。简而言之，这个特性允许我们使用new关键字来创建变量，它将变量自动设置为其默认值。

- bool类型设置为false；
- 数值类型设置为0（如果是浮点数，则设置为0.0）；
- char类型设置为单个空字符；
- BigInteger变量设置为0；
- DateTime类型设置为1/1/0001 12:00:00 AM；
- 对象引用（包括string）设置为null。

说明　稍后将阐述上面列出的BigInteger数据类型。

尽管使用new关键字创建基本数据类型的变量显得很笨重，如下代码却是可行的C#代码：

```
static void NewingDataTypes()
{
  Console.WriteLine("=> Using new to create variables:");
  bool b = new bool();              // 设置为false
  int i = new int();                // 设置为0
  double d = new double();          // 设置为0
  DateTime dt = new DateTime();     // 设置为1/1/0001 12:00:00 AM
  Console.WriteLine("{0}, {1}, {2}, {3}", b, i, d, dt);
  Console.WriteLine();
}
```

3.4.3　数据类型类的层次结构

要注意很有趣的一点是，.NET的基本数据类型都有一个*类层次结构*。如果你刚接触继承的话，可以在第6章中找到详细的内容。在这里，只需要理解处于类层次结构顶端的类型会为派生类型提供一些默认行为。这些核心系统类型之间的关系如图3-2所示。

注意，所有这些类型都派生自System.Object，它定义了一组.NET基础类库中所有类型都具有的方法（如ToString()、Equals()、GetHashCode()等，在第6章中会详细介绍这些方法）。

还要注意，很多数值数据类型都派生自System.ValueType类。派生自ValueType的类型都会自动在栈上进行分配，因此有一个可预测的生命周期，而且非常高效。另一方面，在继承链上没有System.ValueType的类型（如System.Type、System.String、System.Array、System.Exception以及System.Delegate）不会在栈上分配，而是在垃圾回收堆上进行分配。

在此，我们不会过多涉及System.Object和System.ValueType的细节，只需要理解由于C#关键字（如int）只是相应系统类型（这里是System.Int32）的简化符号，并且System.Int32（C#中的int）最终派生自System.Object，也就可以调用它的任何公有成员，如下所示的额外辅助函数可以说明这一点儿：

```
static void ObjectFunctionality()
{
  Console.WriteLine("=> System.Object Functionality:");

  // C# int是System.Int32的简化，它继承了System.Object的如下成员
  Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
  Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
  Console.WriteLine("12.ToString() = {0}", 12.ToString());
  Console.WriteLine("12.GetType() = {0}", 12.GetType());
  Console.WriteLine();
}
```

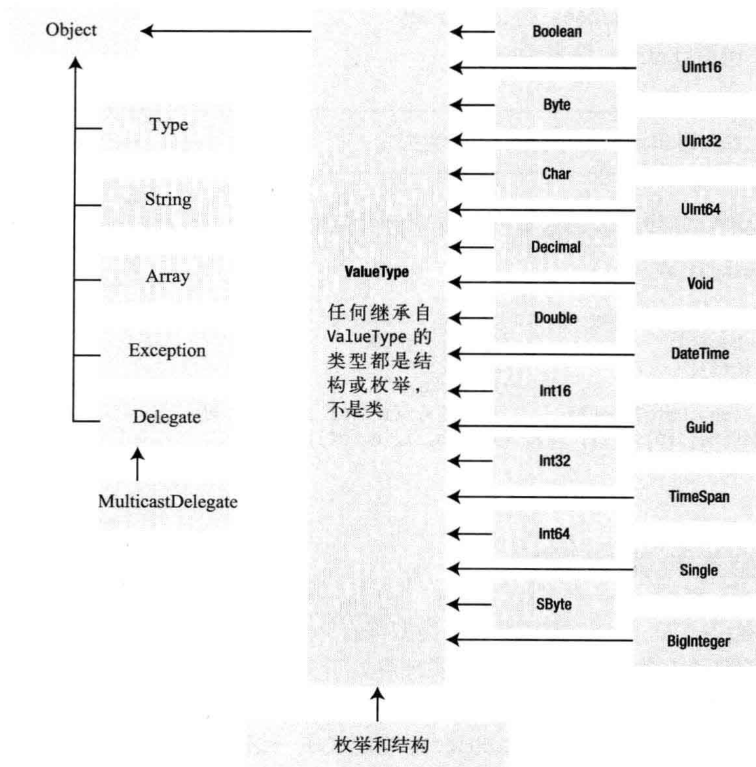


图3-2　系统类型的类层次结构

如果在Main()方法中调用这个方法，就会得到如下所示的输出结果。

```
=> System.Object Functionality:

12.GetHashCode() = 12
12.Equals(23) = False
12.ToString() = 12
12.GetType() = System.Int32
```

3.4.4 数值数据类型的成员

为了继续研究内建的C#数据类型，必须知道.NET的数值类型支持MaxValue和MinValue属性，这两个属性说明了给定的类型可以存储的范围。除了MinValue/MaxValue属性之外，一个给定的系统数据类型还可能定义其他更有用的成员。例如，使用System.Double类型可以获取最小正数和无穷大数（这些可能是喜爱数学的你所感兴趣的）。看看下面的辅助方法：


```
static void DataTypeFunctionality()
{
  Console.WriteLine("=> Data type Functionality:");

  Console.WriteLine("Max of int: {0}", int.MaxValue);
  Console.WriteLine("Min of int: {0}", int.MinValue);
  Console.WriteLine("Max of double: {0}", double.MaxValue);
  Console.WriteLine("Min of double: {0}", double.MinValue);
  Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
  Console.WriteLine("double.PositiveInfinity: {0}",
    double.PositiveInfinity);
  Console.WriteLine("double.NegativeInfinity: {0}",
    double.NegativeInfinity);
  Console.WriteLine();
}
```

3.4.5 System.Boolean的成员

下面考虑System.Boolean数据类型。C#的bool类型的值只能来自集合{true|false}。基于这一点，应该明确System.Boolean不支持MinValue/MaxValue属性集合，但是支持TrueString/ FalseString属性集合（相应地返回True或False）。将以下语句添加到DataTypeFunctionality()辅助方法中：

```
Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
Console.WriteLine("bool.TrueString: {0}", bool.TrueString);
```

3.4.6 System.Char的成员

C#的文本数据是由string和char关键字表示的，它们是System.String和System.Char的简化符号，二者都是基于Unicode的。string表示一组连续的字符（如"Hello"），而char则表示string类型中的单个字符（如'H'）。

System.Char类型除了保存一个字符数据之外，还提供了大量的功能。使用System.Char的静态方法，可以判定一个给定的字符是否是数字、字母、标点符号或其他。看一下下面的代码：

```
static void CharFunctionality()
{
  Console.WriteLine("=> char type Functionality:");

  char myChar = 'a';
  Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
  Console.WriteLine("char.IsLetter('a'): {0}", char.IsLetter(myChar));
  Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
    char.IsWhiteSpace("Hello There", 5));
  Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
    char.IsWhiteSpace("Hello There", 6));
```

```
    Console.WriteLine("char.IsPunctuation('?'): {0}",
      char.IsPunctuation('?'));
    Console.WriteLine();
  }
```

可见，这些System.Char的静态成员中的每一个都有两个调用约定：一个字符或是一个字符串，该字符串带一个指定要测试的字符所在位置的数值索引。

3.4.7 从字符串数据中解析数值

我们也知道.NET数据类型提供了一种能力，即通过给定文本生成（例如解析）相应的底层类型的变量。这个技术在想把用户输入的数据（例如基于GUI的下拉列表中的选择）转换成一个数值时非常有用。考虑下面ParseFromStrings()方法中的解析逻辑：

```
static void ParseFromStrings()
{
  Console.WriteLine("=> Data type parsing:");

  bool b = bool.Parse("True");
  Console.WriteLine("Value of b: {0}", b);
  double d = double.Parse("99.884");
  Console.WriteLine("Value of d: {0}", d);
  int i = int.Parse("8");
  Console.WriteLine("Value of i: {0}", i);
  char c = Char.Parse("w");
  Console.WriteLine("Value of c: {0}", c);
  Console.WriteLine();
}
```

3.4.8 System.DateTime和System.TimeSpan

System命名空间定义了很多有用的数据类型，对于这些类型，没有C#关键字，比如DateTime和TimeSpan结构（对于图3-2所示的System.Guid和System.Void的研究就留给感兴趣的读者，但要注意System命名空间中的这两种数据类型在多数应用程序中基本不用）。

DateTime类型包含了表示某个日期（月、日、年）的数据以及时间值，可以使用指定的成员以各种形式将它们格式化。TimeSpan结构允许你方便地使用各个成员定义和转换时间单位：

```
static void UseDatesAndTimes()
{
  Console.WriteLine("=> Dates and Times:");

  // 这个构造函数接受年、月、日
  DateTime dt = new DateTime(2011, 10, 17);

  // 它是一个月中的哪一天
  Console.WriteLine("The day of {0} is {1}", dt.Date, dt.DayOfWeek);

  // 月份现在是12月
  dt = dt.AddMonths(2);
  Console.WriteLine("Daylight savings: {0}", dt.IsDaylightSavingTime());

  // 构造函数接受小时、分钟和秒
  TimeSpan ts = new TimeSpan(4, 30, 0);
```

```
    Console.WriteLine(ts);

    // 从当前TimeSpan减去15分钟并且输出结果
    Console.WriteLine(ts.Subtract(new TimeSpan(0, 15, 0)));
  }
```

3.4.9 System.Numerics.dll程序集

System.Numerics命名空间定义了BigInteger结构。顾名思义，BigInteger数据类型可用来表示极大的数值，它没有固定的上下限。

> **说明** System.Numerics命名空间还定义了一个Complex结构，用来对数学中的复数数据进行建模（如虚部数据、实部数据、双曲正切数）。如果感兴趣的话，可以查阅.NET Framework 4.5 SDK文档。

尽管大多数.NET应用程序都不需要使用BigInteger结构，但一旦需要定义较大的数值时，你要做的第一件事就是为项目添加System.Numerics.dll程序集的引用。对于当前示例来说，需要执行以下几个步骤。

(1) 在Visual Studio中选择Project → Add Reference...菜单选项。

(2) 在当前库的列表中找到并选择System.Numerics.dll程序集。

(3) 单击Add和Close按钮。

然后，在文件中添加如下的using指令，此时将使用BigInteger数据类型：

```
// BigInteger在这里面
using System.Numerics;
```

此时就可以使用new操作符来创建BigInteger变量了。在构造函数中，可以指定包含浮点数据在内的数值。但是，当你定义一个整数时（如500），运行时将其默认设为int数据类型。同样，浮点数据（如55.333）将默认设为double类型。那么，如何将BigInteger设置为大数，才能使那些用于表示原始数值的默认数据类型不会溢出呢？

最简单的方式是将大数值作为文本字面量，使用静态的Parse()方法将其转换为BigInteger变量。如果需要，还可以向BigInteger类的构造函数直接传递字节数组。

> **说明** 由于BigInteger是不可变的（immutable），因此一旦设置了BigInteger变量的值，就不能改变它了。但是，BigInteger类定义了大量方法，根据对数据的修改返回新的BigInteger对象（例如下面代码中的静态方法Multiply()）。

定义了BigInteger变量之后，你会发现该类的成员与其他固有的C#数据类型（如float、int）的成员十分类似。此外，BigInteger还定义了一些静态成员，用来对BigInteger变量应用基本的数学表达式（如加法和乘法）。下面是使用BigInteger类的示例：

```
static void UseBigInteger()
{
  Console.WriteLine("=> Use BigInteger:");
  BigInteger biggy =
```

```
    BigInteger.Parse("9999999999999999999999999999999999999999999999");
  Console.WriteLine("Value of biggy is {0}", biggy);
  Console.WriteLine("Is biggy an even value?: {0}", biggy.IsEven);
  Console.WriteLine("Is biggy a power of two?: {0}", biggy.IsPowerOfTwo);
  BigInteger reallyBig = BigInteger.Multiply(biggy,
    BigInteger.Parse("888888888888888888888888888888888888888888888"));
  Console.WriteLine("Value of reallyBig is {0}", reallyBig);
}
```

同样重要的是，BigInteger数据类型能够响应C#基本的数学操作符，如+、-和*。因此，在对两个大数执行乘法运算时，不必调用BigInteger.Multiply()，而只需使用下面的代码：

```
BigInteger reallyBig2 = biggy * reallyBig;
```

现在，我希望你能了解的是，对于每一个表示基本数据类型的C#关键字，.NET基础类库中都有一个类型与之对应，每个类型都有其固定的功能。虽然我没有详细介绍这些数据类型的各个成员，但如果感兴趣的话，你可以详细了解它们。关于不同.NET数据类型的详细内容，可以参考.NET Framework 4.5 SDK文档——你肯定会惊叹如此数量的内置功能。

源代码　BasicDataTypes项目的源代码位于Chapter 3子目录下。

3.5 使用字符串数据

System.String提供了很多你原来期望工具类提供的方法，包括返回字符数据长度、查找当前字符串中子字符串、转换大小写等方法。表3-5列出了一些（但不是全部）常见成员。

表3-5　System.String的部分成员

字符串成员	作　用
Length	这个属性返回当前字符串的长度
Compare()	这个方法比较两个字符串
Contains()	这个方法用于判定当前字符串是否包含一个指定的子字符串
Equals()	这个方法测试两个字符串对象是否含有同样的字符数据
Format()	这个静态方法使用本章前面分析过的其他基本类型（如数值数据和其他字符串）和{0}符号以格式化一个字符串
Insert()	这个方法用来将一个字符串插入到给定字符串中
PadLeft() PadRight()	这两个方法用来在字符串内填充字符
Remove() Replace()	使用这些方法来接收一个带有修改（被删除或替换的字符）的字符串的副本
Split()	这个方法返回的String数组包含这个实例中由指定的Char或String数组的元素分隔的子字符串
Trim()	这个方法从当前字符串的头部和尾部移除所有出现的一组指定字符
ToUpper() ToLower()	这两个方法创建一个给定字符串的大写或小写副本

3.5.1 基本的字符串操作

就像我们期望的那样，使用System.String的成员只需要创建字符串数据类型，然后通过点操作符使用所提供的功能。要知道，有些System.String的成员都是静态成员，也因此在类级别（而不是对象级别）进行调用。假设我们新建了一个名为FunWithStrings的控制台应用程序，编写如下方法，并在Main()中进行调用：

```
static void BasicStringFunctionality()
{
  Console.WriteLine("=> Basic String functionality:");
  string firstName = "Freddy";
  Console.WriteLine("Value of firstName: {0}", firstName);
  Console.WriteLine("firstName has {0} characters.", firstName.Length);
  Console.WriteLine("firstName in uppercase: {0}", firstName.ToUpper());
  Console.WriteLine("firstName in lowercase: {0}", firstName.ToLower());
  Console.WriteLine("firstName contains the letter y?: {0}",
    firstName.Contains("y"));
  Console.WriteLine("firstName after replace: {0}", firstName.Replace("dy", ""));
  Console.WriteLine();
}
```

这里不多说什么了，因为这个方法只是调用了字符串本地变量上的各种成员（如ToUpper()、Contains()等）来产生各种格式和转换。下面是初始输出。

```
 ***** Fun with Strings *****

=> Basic String functionality:
Value of firstName: Freddy
firstName has 6 characters.
firstName in uppercase: FREDDY
firstName in lowercase: freddy
firstName contains the letter y?: True
firstName after replace: Fred
```

虽然这些输出结果并不意外，但Replace()方法的调用结果可能会让人误解。实际上，firstName变量根本没有改变。我们得到的是修改后的新的string。稍后，我们将重温字符串的这种不可变性。

3.5.2 字符串拼接

字符串变量可以通过C#的+操作符连接在一起来构建一个更大的字符串变量。你可能知道，这项技术正式的名称是字符串拼接（string concatenation）。下面来看一下新的辅助函数：

```
static void StringConcatenation()
{
  Console.WriteLine("=> String concatenation:");
  string s1 = "Programming the ";
  string s2 = "PsychoDrill (PTP)";
  string s3 = s1 + s2;
  Console.WriteLine(s3);
  Console.WriteLine();
}
```

C#的+符号会被编译器处理为对String.Concat()方法的调用。因此，也可以直接调用String.Concat()来进行字符串拼接操作（尽管这么做确实没有什么好处——要多敲几次键盘）：

```
static void StringConcatenation()
{
  Console.WriteLine("=> String concatenation:");
  string s1 = "Programming the ";
  string s2 = "PsychoDrill (PTP)";
  string s3 = String.Concat(s1, s2);
  Console.WriteLine(s3);
  Console.WriteLine();
}
```

3.5.3 转义字符

与其他基于C的语言一样，在C#字符串字面量中可以包含各种转义字符，用来限制字符数据应该怎样被输到输出流中。每个转义字符都以一个反斜线开始，后跟一个特殊的标记。也许你对这些转义字符的意义不太了解，表3-6列出了最常用的选项。

表3-6　字符串字面量的转义字符

字　符	作　用
\'	将一个单引号插入字符串字面量
\"	将一个双引号插入字符串字面量
\\	将一个反斜线插入字符串字面量。这在定义文件或网络路径时很有用
\a	触发一个系统警报（蜂鸣）。对控制台应用程序来说，这能给用户提供一个声音提示
\n	换行（在Windows平台上）
\r	回车
\t	将一个水平制表符插入字符串字面量

例如，为了输出在每一个单词间都包含一个制表符的字符串，可以使用\t转义字符。假设想创建一个包含引号的字符串字面量，一个定义目录路径的字符串字面量以及一个在输出字符数据后插入3个空行的字符串字面量。为了这样做且不引起编译器错误，需要使用\"、\\和\n转义字符。同样，为了干扰半径为10英尺的范围内的所有人，我在每一个字符串字面量中都嵌入了一个警报（触发一声蜂鸣）。参考如下代码：

```
static void EscapeChars()
{
  Console.WriteLine("=> Escape characters:\a");
  string strWithTabs = "Model\tColor\tSpeed\tPet Name\a ";
  Console.WriteLine(strWithTabs);

  Console.WriteLine("Everyone loves \"Hello World\"\a ");
  Console.WriteLine("C:\\MyApp\\bin\\Debug\a ");

  // 添加4个空行，然后发出一声蜂鸣
  Console.WriteLine("All finished.\n\n\n\a ");
  Console.WriteLine();
}
```

3.5.4 定义逐字字符串

C#引入了以@为前缀的字符串字面量记法，术语称为*逐字字符串*（verbatim string）。使用逐字字符串，可以使对一个字面量的转义字符的处理失效并输出字符串。这在使用表示目录和网络路径的字符串时最有用。因此，不需要使用\\转义字符，可以简单地按如下写代码：

```
// 下面的字符串被逐字输出，所有的转义字符都被显示出来了
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

还要注意逐字字符串可以用来为持续很多行的字符串保留空格：

```
// 使用逐字字符串，空格被保留了
string myLongString = @"This is a very
     very
          very
               long string";
Console.WriteLine(myLongString);
```

还可以通过重复"标记向一个字面量字符串插入一个双引号：

```
Console.WriteLine(@"Cerebus said ""Darrr! Pret-ty sun-sets""");
```

3.5.5 字符串和相等性

在第4章中我们会详细介绍，*引用类型*是在垃圾回收托管堆上分配的对象。默认情况下，当我们对引用类型进行相等性测试（通过C# ==和!=操作符）时，如果引用类型指向内存中的相同对象，则返回true。然而，尽管String数据类型确实是引用类型，但是相等性操作符已经被重定义为比较字符串对象的值，而不是内存中它们引用的对象：

```
static void StringEquality()
{
  Console.WriteLine("=> String equality:");
  string s1 = "Hello!";
  string s2 = "Yo!";
  Console.WriteLine("s1 = {0}", s1);
  Console.WriteLine("s2 = {0}", s2);
  Console.WriteLine();

  // 测试这些字符串的相等性
  Console.WriteLine("s1 == s2: {0}", s1 == s2);
  Console.WriteLine("s1 == Hello!: {0}", s1 == "Hello!");
  Console.WriteLine("s1 == HELLO!: {0}", s1 == "HELLO!");
  Console.WriteLine("s1 == hello!: {0}", s1 == "hello!");
  Console.WriteLine("s1.Equals(s2): {0}", s1.Equals(s2));
  Console.WriteLine("Yo.Equals(s2): {0}", "Yo!".Equals(s2));
  Console.WriteLine();
}
```

注意，C#相等性操作符进行的是区分大小写、逐字符的相等性测试。因此，"Hello!"、"HELLO!"和"hello!"都不相等。同样，始终记住string和System.String之间的联系，要指出的是我们可以使用String的Equals()方法或内嵌的相等性操作符进行相等性测试。最后，因为每一个字符串字面量（如"Yo"）都是有效的System.String实例，所以我们可以从一个固定的字符序列来访问字符串相关的功能。

3.5.6 字符串是不可变的

System.String一个有趣的方面是，一旦将初始值赋给字符串对象，字符数据就不能改变了。乍一看，这可能像是一个明显的谎言。因为我们总是给字符串赋新值，而且System.String类型也定义了许多用于以各种方式（大写、小写等）修改字符数据的方法。然而，如果细究背后发生的事情，就会注意到string类型的方法其实返回了一个修改格式的新字符串对象。

```
static void StringsAreImmutable()
{
  // 设置初始字符串值
  string s1 = "This is my string.";
  Console.WriteLine("s1 = {0}", s1);

  // 大写s1?
  string upperString = s1.ToUpper();
  Console.WriteLine("upperString = {0}", upperString);

  // 不! s1还是同样的格式
  Console.WriteLine("s1 = {0}", s1);
}
```

查看上述代码相关的输出，就可以验证原来的string对象（s1）在调用ToUpper()之后没有变成大写，而是返回修改格式的字符串副本，如下所示。

```
s1 = This is my string.
upperString = THIS IS MY STRING.
s1 = This is my string.
```

如果我们使用C#赋值操作符，同样的不变性命题还是成立的。StringsAreImmutable2()方法的实现过程可以说明这个问题：

```
static void StringAreImmutable2()
{
  string s2 = "My other string";
  s2 = "New string value";
}
```

现在，编译应用程序，然后把程序集加载到ildasm.exe中（参见第1章）。下面的输出显示了为StringsAreImmutable2()方法生成CIL代码后的结果。

```
.method private hidebysig static void  StringsAreImmutable2() cil managed
{
  // 代码大小       14 (0xe)
  .maxstack  1
  .locals init ([0] string s2)
  IL_0000:  nop
  IL_0001:  ldstr      "My other string"
  IL_0006:  stloc.0
  IL_0007:  ldstr      "New string value"
  IL_000c:  stloc.0
  IL_000d:  ret
} // 结束 Program::StringAreImmutable2方法
```

尽管我们还没有研究CIL底层的一些细节，但是要注意到Main()方法多次调用了ldstr（加载字符串）操作码。简而言之，CIL的ldstr操作码在托管堆上加载了一个新的字符串对象。之前包含值"My other string"的字符串对象最终会被垃圾回收。

那么，我们能从中得出什么结论呢？简单来说，string类如果被滥用，它就会变得低效，并导致代码膨胀，特别是进行字符串拼接的时候。如果我们需要表示在应用程序中用到的基本字符数据，如美国社会安全号码（SSN）、姓名或简单字符串文本等，string类就是很好的选择。

然而，如果我们正在构建使用大量文本数据的应用程序（如字处理程序），那么使用String对象来表示字处理数据就是一个糟糕的主意，因为我们最后很可能（通常是间接地）产生字符串数据不必要的副本。那么程序员该怎么做呢？问得好。

3.5.7 System.Text.StringBuilder类型

如果随便弃用的话，String类型可能会很低效，因此.NET基础类库提供了System.Text命名空间。在这个（相对较小的）命名空间中有一个叫做StringBuilder的类。和System.String类相似，StringBuilder定义了很多用来替换或格式化片段的方法。如果我们希望在C#代码文件中使用这个类型，第一步就是导入正确的命名空间（对于一个新的Visual Studio项目，这一步应该已经完成了）：

```
// StringBuilder在这里
using System.Text;
```

StringBuilder的独特之处在于，当我们调用这个类型的成员时，都是直接修改对象内部的字符数据（因此更高效），而不是获取修改后格式的数据副本。当创建StringBuilder实例时，可以通过其中一个构造函数来提供对象的初始值。如果你对构造函数不是很熟悉，可以理解为构造函数允许我们在使用new关键字时使用初始状态来创建对象。下面看一下StringBuilder的用法：

```
static void FunWithStringBuilder()
{
  Console.WriteLine("=> Using the StringBuilder:");
  StringBuilder sb = new StringBuilder("**** Fantastic Games ****");
  sb.Append("\n");
  sb.AppendLine("Half Life");
  sb.AppendLine("Morrowind");
  sb.AppendLine("Deus Ex "+" 2");
  sb.AppendLine("System Shock");
  Console.WriteLine(sb.ToString());

  sb.Replace("2", "Invisible War");
  Console.WriteLine(sb.ToString());
  Console.WriteLine("sb has {0} chars.", sb.Length);
  Console.WriteLine();
}
```

在这里，我们构建了一个StringBuilder，并且将初始值设置为"**** Fantastic Games ****"。可以看到，我们向内部缓冲区追加数据，并且可以随意替换（或移除）字符。默认情况下，StringBuilder只能保存16个字符以下的字符串（如果需要，可以自动扩展），然而，我们可以通过其他构造函数参数来改变这个初始值。

```
// 创建一个初始大小为256的StringBuilder
StringBuilder sb = new StringBuilder("**** Fantastic Games ****", 256);
```

如果追加的字符数超过规定的限制，StringBuilder对象会将它的数据复制到新的实例中，并根据规定的限制来扩大缓冲区。

源代码 FunWithStrings项目的源代码位于Chapter 3子目录下。

3.6 窄化和宽化数据类型转换

既然我们理解了如何和内建的C#数据类型进行交互，下面就研究一下*数据类型转换*的相关主题。假设我们有一个新的控制台应用程序TypeConversions，并定义了如下的类：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with type conversions *****");

    // 把两个short数据相加并输出结果
    short numb1 = 9, numb2 = 10;
    Console.WriteLine("{0} + {1} = {2}",
      numb1, numb2, Add(numb1, numb2));
    Console.ReadLine();
  }

  Static int Add(int x, int y)
  {
    Return x + y;
  }
}
```

注意，Add()方法期望传入两个int参数。然而，Main()方法实际上传入了两个short变量。虽然这可能看上去绝对是不匹配的数据类型，但是程序编译和执行都没有出错，并返回了我们期望的结果19。

因为不可能丢失数据，所以编译器认为这段代码从语法上说是正确的。由于short的最大值（32 767）在int范围（2 147 483 647）内，编译器就把每一个short隐式宽化为int。正式地说，宽化用于定义隐式向上转换，并且不会导致数据丢失。

说明 如果希望查看每一个C#数据类型允许的宽化（和窄化）转换，请在.NET Framework 4.5 SDK文档中查找Type Conversion Tables（类型转换表）的主题。

尽管这个隐式宽化在之前的示例中工作得很好，但在其他时候这个“特性”可能会引起编译时错误。例如，假设我们已经为numb1和numb2设置了（加在一起时）会溢出short最大值的值。同样，假设我们把Add()方法的返回值保存在新的short本地变量中，而不是直接把结果输出到控制台：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with type conversions *****");

  // 下面会有编译器错误
```

```
    short numb1 = 30000, numb2 = 30000;
    short answer = Add(numb1, numb2);

    Console.WriteLine("{0} + {1} = {2}",
      numb1, numb2, answer);
    Console.ReadLine();
  }
```

在这里，编译器报告了如下的错误：


```
Cannot implicitly convert type 'int' to 'short'. An explicit conversion exists (are you missing a cast?)
```

问题是，尽管Add()方法可以返回int的值60 000（因为它在System.Int32的范围内），但是值不能保存在short中（因为它溢出了这个数据类型的边界）。正式地说，CLR不能应用窄化运算。窄化和宽化是相对的，因为大值保存在小变量中。

要指出很重要的一点：即使我们可以推断出窄化转换会成功，所有的窄化转换也会导致编译器错误。例如，如下代码会导致编译器错误：

```
// 另外一个编译器错误
static void NarrowingAttempt()
{
  byte myByte = 0;
  int myInt = 200;
  myByte = myInt;

  Console.WriteLine("Value of myByte: {0}", myByte);
}
```

在这里，保存在int变量（myInt）中的值肯定位于byte范围内，因此我们期望这样的窄化转换不会导致运行时错误。然而，由于C#是类型安全的，我们也确实会收到编译器错误。

如果希望通知编译器我们想要处理窄化运算引起的可能的数据丢失，就必须使用C#强制转换操作符()来进行显式强制转换。下面的代码修改了Program类型。

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with type conversions *****");
    short numb1 = 30000, numb2 = 30000;

    // 显式强制转换int为short（并且允许数据丢失）
    short answer = (short)Add(numb1, numb2);

    Console.WriteLine("{0} + {1} = {2}",
      numb1, numb2, answer);
    NarrowingAttempt();
    Console.ReadLine();
  }

  static int Add(int x, int y)
  {
    return x + y; }
  }
```

```
    static void NarrowingAttempt()
    {
      byte myByte = 0;
      int myInt = 200;

      // 显式强制转换int为byte (没有数据丢失)
      myByte = (byte)myInt;
      Console.WriteLine("Value of myByte: {0}", myByte);
    }
  }
```

这种情况下代码会编译，但添加的结果是完全错误的：

```
***** Fun with type conversions *****
30000 + 30000 = -5536
Value of myByte: 200
```

我们看到，显式强制转换允许我们强制编译器应用窄化转换（即使这样做可能会导致数据丢失）。就NarrowingAttempt()方法而言，这样做没有问题，因为值200处在byte范围内。然而，对于Main()中两个short相加的情况来说，最后的结果完全不能接受（30 000 + 30 000 = –5536?）。

如果我们在构建一个不能接受数据丢失的应用程序，C#提供的checked和unchecked关键字将确保数据丢失肯定会被检测到。

3.6.1 checked关键字

先来看一下checked关键字的用法。假设Program里有一个新的方法要将两个byte相加，且每一个都被赋了一个小于最大值(255)的值。如果我们要将这两个类型的值相加(将返回的int强制转换为byte)，就可以假设结果恰好是所有成员的和：

```
static void ProcessBytes()
{
  byte b1 = 100;
  byte b2 = 250;
  byte sum = (byte)Add(b1, b2);

  // sum应该保存值350。然而，我们得到了值94
  Console.WriteLine("sum = {0}", sum);
}
```

如果我们查看这个应用程序的输出结果，可能会惊讶地发现sum保存了值94（而不是期望的350）。原因很简单，由于System.Byte只能保存0~255之间的值（一共只有256个位置），sum现在保存的是溢出值（350 – 256 = 94）。默认情况下，如果我们不采取纠正措施，发生上溢/下溢的时候就不会给出出错信息。

要在应用程序中处理上溢或下溢的情况，我们有两个选择。第一个选择是利用聪明才智和编程技巧来手动处理所有上溢和下溢的情况。当然，这项技术的问题在于，我们是人不是神，即使最好的尝试都可能导致错误从眼皮底下溜走。

幸好，C#提供了checked关键字。当我们把一个语句（或者语句块）包装在checked关键字域内时，C#编译器会使用额外的CIL指令来测试在将两个数值数据类型相加、相乘、相减和相除时可能产生的溢出情况。

如果发生了溢出，就会得到一个运行时异常：System.OverflowException。第7章会探讨所有结构化异常处理的细节以及try和catch关键字的用法。这里不涉及更多细节，观察如下更新：

```
static void ProcessBytes()
{
  byte b1 = 100;
  byte b2 = 250;

  // 这次告诉编译器增加CIL代码，如果发生上溢或下溢就抛出异常
  try
  {
    byte sum = checked((byte)Add(b1, b2));
    Console.WriteLine("sum = {0}", sum);
  }
  catch (OverflowException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

注意，Add()的返回值被包装在checked关键字的作用域中。由于和大于一个byte，就触发了一个运行时异常。注意下面通过Message属性输出的错误消息。

```
Arithmetic operation resulted in an overflow.
```

如果希望对一段代码语句块进行强制溢出检测，可以按如下所示定义checked作用域：

```
try
{
  checked
  {
    byte sum = (byte)Add(b1, b2);
    Console.WriteLine("sum = {0}", sum);
  }
}
catch (OverflowException ex)
{
  Console.WriteLine(ex.Message);
}
```

在这两个例子中，代码都会自动运算可能的溢出情况，并且在遇到这种情况时发出一个溢出异常。

3.6.2 设定项目级别的溢出检测

如果要创建的应用程序不允许发生任何未知的溢出，我们可能会很烦恼，因为要把无数代码行放到checked关键字作用域中。另外一种可选的方案是，C#编译器提供了/checked标志。如果启用的话，所有的运算都会被检查是否溢出，无需使用C# checked关键字。如果发现溢出，我们会收到一个运行时异常。

要使用Visual Studio启用这个标志，可以打开项目属性页，然后单击Build标签中的Advanced按钮。在结果对话框中选择Check for arithmetic overflow/underflow（检测运算上溢/下溢）复选框（如图3-3所示）。

当我们创建调试编译时，启用这个设置会很有用。一旦代码中所有溢出异常都解决了，你完全可以对之后的编译禁用/checked标志（这样会提升应用程序的运行时性能）。

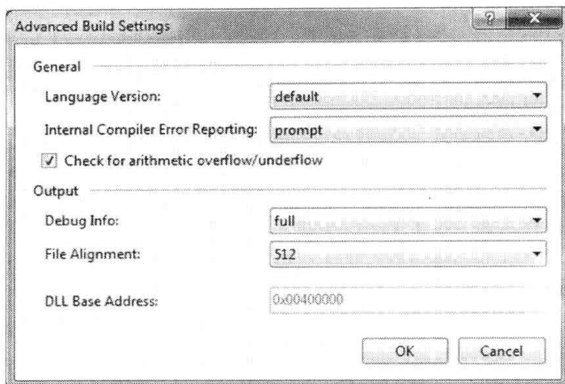


图3-3　开启项目级别的上溢/下溢数据检测

3.6.3　unchecked关键字

现在，假设我们已经启用了项目级别的设置，如果有一段代码中的数据丢失是可以接受的，我们该怎么做呢？由于/checked标志会检查所有运算逻辑，C#提供的unchecked关键字会禁止个别情况下抛出溢出异常。这个关键字的用法和checked关键字的很相似，我们可以指定一个语句或者一个语句块。

```
// 假设启用了/checked，这段代码不会触发运行时异常
unchecked
{
  byte sum = (byte)(b1 + b2);
  Console.WriteLine("sum = {0} ", sum);
}
```

现在总结一下C#的checked和unchecked关键字，记住.NET运行时的默认行为是忽略运算溢出。当我们需要有选择性地处理分散的语句时，可以使用checked关键字。如果希望在整个应用程序中捕捉溢出错误，可以启用/checked标志。最后，如果有一段代码中的溢出是可接受的（因此不应该触发运行时异常），可以使用unchecked关键字。

源代码　TypeConversions项目的源代码位于Chapter 3子目录下。

3.7　隐式类型本地变量

本章到目前为止，我们在定义本地变量时都是显式地指定变量的实际数据类型。

```
static void DeclareExplicitVars()
{
  // 显式类型本地变量的声明如下
  // dataType variableName = initialValue;
  int myInt = 0;
  bool myBool = true;
  string myString = "Time, marches on...";
}
```

尽管为每个变量显式地指定数据类型一直是很好的做法，但C#语言还可以使用var关键字创建隐式类型的本地变量。使用var关键字不必指定具体的数据类型（如int、bool、string）。这么做时，编译器将根据本地数据点的初始值来自动推断实际的数据类型。

为了演示隐式类型的作用，我们创建一个名为ImplicitlyTypedLocalVars的Console Application项目。注意，前面方法中的本地变量，现在可以按如下方式进行声明：

```
static void DeclareImplicitVars()
{
// 隐式类型本地变量的声明如下
// var variableName = initialValue;
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";
}
```

说明 严格地说，var并不是C#关键字。你可以使用“var”作为变量、参数和字段的名字而不会导致编译时错误。尽管如此，当var标记用于数据类型时，编译器根据语境将其视为关键字。

在这种情况下，根据设置的初始值，编译器可以推断myInt实际上是System.Int32类型，myBool是System.Boolean类型，myString是System.String类型。要验证这一点，你可以通过反射来输出类型名称。正如我们将在第15章中详细介绍的那样，反射就是在运行时判定类型的组成。例如，使用反射，你可以判定一个隐式类型本地变量的数据类型。使用如下的代码更新方法：

```
static void DeclareImplicitVars()
{
// 隐式类型本地变量
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";

// 输出实际类型
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
  Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
  Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

说明 要知道的是，你可以用隐式类型表示任何类型，包括数组、泛型类型（见第9章），以及自定义的类型。你还将在本书中看到隐式类型的其他示例。

如果在Main()中调用DeclareImplicitVars()方法，输出结果将如下所示。

```
***** Fun with Implicit Typing *****

myInt is a: Int32
myBool is a: Boolean
myString is a: String
```

3.7.1　隐式类型变量的限制

对于var关键字的使用，自然有很多限制。首先，隐式类型只能用于方法或属性范围内的本地变量。用var关键字定义返回值、参数或自定义类型的字段数据，都是不合法的。例如，下面的类定义将导致多个编译时错误：

```
class ThisWillNeverCompile
{
  // 错误! var不能用于字段数据
  private var myInt = 10;

  // 错误! var不能用于返回值或参数类型
  public var MyMethod(var x, var y){}
}
```

同样，用var关键字声明的本地变量必须在声明时分配初始值，并且这个初始值不能为null。后一个限制是有意义的，因为编译器仅仅根据null无法推断该变量在内存中实际指向的数据类型。

```
// 错误! 必须分配值
var myData;

// 错误! 必须在声明时分配值
var myInt;
myInt = 0;

// 错误! 不能分配null作为初始值
var myObj = null;
```

然而，为隐式类型本地变量分配初始值并进行推断之后（必须是引用类型），就可以对其分配null了。

```
// 没问题，SportsCar是一个引用类型
var myCar = new SportsCar();
myCar = null;
```

此外，可以将隐式类型本地变量的值分配给其他变量（不管它是否为隐式类型）。

```
// 同样没问题
var myInt = 0;
var anotherInt = myInt;

string myString = "Wake up!";
var myData = myString;
```

同样，你还可以向调用方返回一个隐式类型本地变量，只要方法的返回类型与var定义的数据点的实际类型是相同的。

```
static int GetAnInt()
{
  var retVal = 9;
  return retVal;
}
```

3.7.2　隐式类型数据是强类型数据

要非常清楚的是，隐式类型的本地变量是强类型数据。因此，var关键字与脚本语言（如JavaScript、

Perl）所用的技术和COM的Variant数据类型并不相同，对于后两者来说，一个变量在应用程序的生命周期中可以保存不同的类型（通常称为动态类型）。

说明 C#允许C#中的动态类型使用dynamic关键字——你一定非常吃惊！你将在第18章中学习它。

实际上，类型推断延续了C#语言的强类型特性，并且只会在编译时影响变量的声明。之后，该数据点将被视为它声明的类型。为该变量分配不同的类型将导致编译时错误。

```
static void ImplicitTypingIsStrongTyping()
{
   // 编译器知道“s”是一个System.String
   var s = "This variable can only hold string data!";
   s = "This is fine...";

   // 可以调用实际类型的所有成员
   string upper = s.ToUpper();

   // 错误！不能将数值数据分配给字符串
   s = 44;
}
```

3.7.3 隐式类型本地变量的用途

了解了声明隐式类型本地变量的语法后，你肯定会奇怪这个结构有什么用呢？首先，使用var声明本地变量并不能带来什么好处。这样做会给其他阅读代码的人带来困扰，由于你无法快速判断实际的数据类型，因此难以理解变量的整体功能。所以，如果需要int，就应该声明为int。

但是，在第12章开头我们将看到，LINQ技术使用了查询表达式，它可以根据表达式本身的格式产生动态创建的结果集。由于在某些情况下根本无法显式定义查询的返回类型，这时隐式类型就非常有用了。不要担心读不懂下面的LINQ示例代码，你只需要看看是否能指出subset的实际数据类型：

```
static void Linq QueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // LINQ查询
  var subset = from i in numbers where i < 10 select i;

  Console.Write("Values in subset: ");
  foreach (var i in subset)
  {
    Console.Write("{0} ", i);
  }
  Console.WriteLine();

  // 嗯……subset是什么类型呢
  Console.WriteLine("subset is a: {0}", subset.GetType().Name);
  Console.WriteLine("subset is defined in: {0}", subset.GetType().Namespace);
}
```

你可能认为subset的数据类型是整数数组，表面看来好像真是这样，实际上它是一种底层LINQ数据类型。除非长时间接触LINQ，或者直接打开ildasm.exe中的编译图形，否则你肯定不会知道这种

情况。好在你使用LINQ时几乎不需要关注查询返回值的基础类型，只需要将值赋给一个隐式类型的局部变量即可。

事实上，可以说只有在定义LINQ查询的返回数据时才应该使用var关键字。记住，如果需要int，就声明int！在产品代码中滥用隐式类型（通过var关键字）被认为是一种糟糕的设计。

源代码　ImplicitlyTypedLocalVars项目的源代码位于Chapter 3子目录下。

3.8　C#迭代结构

所有的编程语言都提供了重复代码块直到满足终止条件的方式。不管过去你用过什么语言，C#迭代语句都不会让你感到很惊讶，只需一点点介绍你就能了解它。C#提供了如下4个迭代结构：

- for循环
- foreach/in循环
- while循环
- do/while循环

让我们依次快速探讨一下每一个循环结构，使用一个新的名为IterationsAndDecisions的控制台应用程序项目。

说明　我想让最后一节简明扼要，因此这里假定你在C#编程语言中用过类似的关键字（if、for、switch等）。要了解更多信息，请查阅.NET Framework 4.5 SDK文档的主题“Iteration Statements (C# Reference)”、“Jump Statements (C# Reference)”和“Selection Statements (C# Reference)”。

3.8.1　for循环

如果需要迭代一段代码固定次数，for语句会提供很大的灵活性。从本质上说，我们可以指定一段代码重复的次数和终止条件。不啰嗦了，直接给出一段语法示例：

```
//基本循环
static void ForLoopExample()
{
  //注意！“i”只在for循环域内可见
  for(int i = 0; i < 4; i++)
  {
    Console.WriteLine("Number is: {0} ", i);
  }
  //“i”在这里不可见
}
```

所有以前学过的C、C++以及Java的技巧都可用于构建C# for语句。我们可以创建复杂的终止条件，构建无限循环，通过--操作符进行反向循环，使用goto、continue和break关键字。

3.8.2 foreach循环

C# foreach关键字允许遍历某种容器中的所有项，无需测试数组的上限。不过与for循环不同，foreach循环只会以线性(n+1)的方式遍历容器（不能执行向后遍历、每隔三个元素访问一次之类的操作）。

然而，当你只需逐项遍历某个集合时，foreach循环是最佳选择。下面是两个使用foreach的示例，一个遍历字符串数组，另外一个遍历整数数组。注意：in关键字前面的数据类型代表容器中的数据类型：


```
//使用foreach迭代数组项
static void ForEachLoopExample()
{
  string[] carTypes = {"Ford", "BMW", "Yugo", "Honda" };
  foreach (string c in carTypes)
    Console.WriteLine(c);

  int[] myInts = { 10, 20, 30, 40 };
  foreach (int i in myInts)
    Console.WriteLine(i);
}
```

in关键字后面的项可以是个简单的数组（比如这里），更确切地说是任何实现IEnumerable接口的类。学习第9章后你会了解，.NET基础类库配备了大量包含普通抽象数据类型（ADT）实现的集合。这些项中的任意一个（如泛型List<T>）都可用在foreach循环中。

在foreach结构中使用隐式类型

在foreach循环结构中也可以使用隐式类型。如你所愿，编译器可以准确地推断“类型的类型”。回想一下本章前面讲过的LINQ示例方法。由于我们不知道subset变量具体的基础数据类型，可以使用隐式类型迭代结果集：

```
static void LinqQueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // LINQ 查询!
  var subset = from i in numbers where i < 10 select i;

  Console.Write("Values in subset: ");
  foreach (var i in subset)
  {
    Console.Write("{0} ", i);
  }
}
```

3.8.3 while和do/while循环结构

当希望执行一段语句直到满足某个终止条件时，while循环结构很有用。在while循环的作用域内，我们当然要确保终止事件会产生，否则就会产生无限循环。在如下代码中，“In while loop”消息会不断输出，直到用户在命令提示符中输入yes才结束循环。

```
static void WhileLoopExample()
{
  string userIsDone = "";

  //对字符串的小写副本进行测试
```

```
    while(userIsDone.ToLower() != "yes")
    {
      Console.WriteLine("In while loop");
      Console.Write("Are you done? [yes] [no]: ");
      userIsDone = Console.ReadLine();
    }
  }
```

和while循环密切相关的是do/while语句。和简单的while循环相似，当我们需要进行一些次数不定的动作时，可以使用do/while。区别是do/while循环肯定会执行至少一次对应的代码块。相比之下，如果一开始终止条件就是false，很可能while循环就不会执行。

```
  static void DoWhileLoopExample()
  {
    string userIsDone = "";

    do
    {
      Console.WriteLine("In do/while loop");
      Console.Write("Are you done? [yes] [no]: ");
      userIsDone = Console.ReadLine();
    }while(userIsDone.ToLower() != "yes"); // 注意分号
  }
```

3.9　条件结构和关系/相等操作符

既然我们可以对一段语句进行迭代，下一个相关概念就是如何控制程序执行的流程。C#定义了两个结构来根据各种情况改变程序的流程：

- if/else语句
- switch语句

3.9.1　if/else语句

首先是if/else语句。然而，和C、C++中的不同，C#中的if/else语句只能作用于布尔表达式，不能用于诸如-1和0这样的值。

3.9.2　关系/相等操作符

if/else语句通常会包含表3-7列出的一些C#操作符，用来获取文本的布尔值。

表3-7　C#关系和相等操作符

C#关系/相等操作符	使用示例	作　用
==	if(age == 30)	如果每一个表达式都相同，就返回true
!=	if("Foo" != myStr)	如果每一个表达式都不同，就返回true
< > <= >=	if(bonus < 2000) if(bonus > 2000) if(bonus <= 2000) if(bonus >= 2000)	如果表达式A(bonus)小于、大于、小于等于或大于等于表达式B(2000)，就返回true

再次提醒，C和C++程序员需要知道，用不等于0的值来测试条件的老技巧不适用于C#。假设你希望看看一个正在操作的字符串是不是超过0个字符，你可能会这些写：

```
static void IfElseExample()
{
  //这是不合法的，因为Length返回的是int而不是bool
  string stringData = "My textual data";
  if(stringData.Length)
  {
    Console.WriteLine("string is greater than 0 characters");
  }
}
```

如果我们希望使用String.Length属性来检测真或假，就必须修改条件表达式，将它解析为一个布尔值。

```
//合法的，它解析为true或false
if(stringData.Length > 0)
{
  Console.WriteLine("string is greater than 0 characters");
}
```

3.9.3 逻辑操作符

一个if语句也可能会由复杂的表达式组成，并且还可以包含else语句来执行更复杂的测试，其语法和C（++）以及Java很相似。如表3-8所示，C#有一组逻辑操作符用于构建复杂的表达式。

表3-8 C#逻辑操作符

逻辑操作符	示 例	作 用
&&	if(age == 30 && name == "Fred")	逻辑与操作符。如果所有表达式均为true，则返回true
||	if(age == 30 || name == "Fred")	逻辑或操作符。只要一个表达式为true，则返回true
!	if(!myBool)	逻辑非操作符。如果表达式为false，则返回true；如果表达式为true，则返回false

说明 &&和||操作符在必要时都会“短路”。也就是说，如果一个表达式被确定为false，其他的表达式就不会被检查了；要想检查所有的表达式，可以使用相关的&和|操作符。

3.9.4 switch语句

C#提供的另外一个简单的选择结构就是switch语句。和其他C系列语言中的一样，switch语句允许我们根据预定义的选择来处理程序流程。例如，Main()逻辑根据两个可能选项中的一个来输出某个字符串消息（default用来处理无效的选择），如下所示：

```
//根据数值进行转换
static void SwitchExample()
{
```

```
    Console.WriteLine("1 [C#], 2 [VB]");
    Console.Write("Please pick your language preference: ");

    string langChoice = Console.ReadLine();
    int n = int.Parse(langChoice);

    switch (n)
    {
      case 1:
        Console.WriteLine("Good choice, C# is a fine language.");
      break;
      case 2:
        Console.WriteLine("VB: OOP, multithreading, and more!");
      break;
      default:
        Console.WriteLine("Well...good luck with that!");
      break;
    }
  }
```

说明　C#要求所有情况（包括default）包含以break或goto终止的可执行语句来避免失败。

C# switch语句一个不错的特性是，除了数值数据之外，我们还可以计算字符串数据。下面这个更新后的switch语句完成了这个特别的任务（注意，如果使用这种方法的话，就不需要把用户数据转换为数值）：

```
static void SwitchOnStringExample()
{
  Console.WriteLine("C# or VB");
  Console.Write("Please pick your language preference: ");

  string langChoice = Console.ReadLine();
  switch (langChoice)
  {
    case "C#":
      Console.WriteLine("Good choice, C# is a fine language.");
    break;
    case "VB":
      Console.WriteLine("VB: OOP, multithreading and more!");
    break;
    default:
      Console.WriteLine("Well...good luck with that!");
    break;
  }
}
```

还可以对枚举数据类型使用switch。你会在第4章看到C# enum关键字可以定义一组名/值对。作为尝鲜，我们来看最后一个辅助方法，它对System.DayOfWeek枚举执行switch检测。你会看到一些还没有介绍过的语法，请先把注意力放在检查枚举本身上。其他部分将在后面的章节一一介绍。

```
static void SwitchOnEnumExample()
{
  Console.Write("Enter your favorite day of the week: ");
  DayOfWeek favDay;

  try
```

```
    {
      favDay = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), Console.ReadLine());
    }
    catch (Exception)
    {
      Console.WriteLine("Bad input!");
      return;
    }

    switch (favDay)
    {
      case DayOfWeek.Friday:
        Console.WriteLine("Yes, Friday rules!");
      break;
      case DayOfWeek.Monday:
        Console.WriteLine("Another day, another dollar");
      break;
      case DayOfWeek.Saturday:
        Console.WriteLine("Great day indeed.");
      break;
      case DayOfWeek.Sunday:
        Console.WriteLine("Football!!");
      break;
      case DayOfWeek.Thursday:
        Console.WriteLine("Almost Friday...");
      break;
      case DayOfWeek.Tuesday:
        Console.WriteLine("At least it is not Monday");
      break;
      case DayOfWeek.Wednesday:
        Console.WriteLine("A fine day.");
      break;
    }
  }
```

源代码 IterationsAndDecisions项目的源代码位于Chapter 3子目录下。

3.10 小结

本章解析了C#编程语言的各个核心方面。在这里，我们研究了在构建任何应用程序时我们都会感兴趣的一些常见的结构。在研究了应用程序对象的作用之后，我们又了解到每一个C#可执行程序都必须有一个类型，用来定义作为程序入口点的Main()方法。在Main()方法内，我们创建了很多赋予应用程序生命的对象。

接着，我们深入探讨了C#内建数据类型的一些细节，并且了解到每一个数据类型关键字（如int）实际上是System命名空间中完整类型（这里就是System.Int32）的简化符号。沿着这个思路，我们还学习了宽化和窄化的作用以及checked和unchecked关键字的作用。

再接下来，我们介绍了用var关键字创建的隐式类型的作用。正如我们讨论的那样，隐式类型的大多数使用场景都与LINQ编程模型有关。最后，我们快速介绍了C#支持的各种迭代和条件结构。

我们对基础细节已经有了一定的了解，下一章将完成对核心语言特性的研究。此后，你就可以学习C#的面向对象特性了。

第4章

C#核心编程结构 II

本章的内容是前一章的补充，我们会在本章结束对C#编程语言核心方面的研究。首先介绍有关构造C#方法的各种细节，然后探讨out、ref和params关键字。同时，我们还将研究可选参数和命名参数的作用。

在研究完方法重载的主题后，下一个任务就是研究使用C#语法操作数组类型的细节，并了解相关的System.Array类类型中包含的功能。

除此之外，本章还讨论了枚举结构以及结构类型，并详细介绍了值类型和引用类型之间的区别。最后探讨了可空数据类型以及?和??操作符的作用。读完本章之后，你就可以很好地学习从第5章开始介绍的C#面向对象知识。

4.1 方法和参数修饰符

我们首先研究定义类型方法的一些细节。和Main()方法（见第3章）相似，自定义方法可以有或没有参数，也可以有或没有返回值。在之后的几章中我们会看到，方法可以在类或结构的作用域内实现（还可以在接口类型中设置原型）、并且可以被各种关键字（如static、virtual、public、new等）修饰以限制其行为。在这里，每一个方法都会遵循下面的基本格式：

```
// 还记得吗？静态方法可以被直接调用，无需创建类的实例
class Program
{
  // static返回值类型 方法名(参数 列表) {/*实现*/}
  static int Add(int x, int y){ return x + y; }
}
```

虽然在C#中方法的定义简单明了，但还是有一些我们可以用来控制参数如何传入目标方法的关键字，如表4-1所示。

表4-1 C#参数修饰符

参数修饰符	作　用
（无）	如果一个参数没有用参数修饰符标记，则认为它将按值传递（pass by value），这意味着被调用的方法收到原始数据的一份副本
out	输出参数由被调用的方法赋值，因此它按引用传递（pass by reference）。如果被调用的方法没有给输出参数赋值，就会出现编译器错误
ref	调用者赋初值，并且可以由被调用的方法可选地重新赋值（因为数据是按引用传递的）。如果被调用的方法未能给ref参数赋值，也不会有编译器错误

（续）

参数修饰符	作　用
params	这个参数修饰符允许将一组可变数量的参数作为单独的逻辑参数进行传递。方法只能有一个parms修饰符，而且必须是方法的最后一个参数。事实上，你不会经常使用params修饰符。但要知道的是，基础类库中的许多方法都使用了这个C#语言特性

为了说明这些关键字的使用，新建一个控制台应用程序FunWithMethods。现在，让我们依次浏览每一个关键字的作用。

4.1.1 默认的参数传递行为

参数传入函数的默认行为是按值传递。简单来说，如果没有为参数标记参数相关的修饰符，数据的副本就会被传入函数。本章的末尾会说明，到底复制什么取决于参数是值类型还是引用类型。而现在，假设Program类中的如下方法操作两个按值传递的数值数据类型：

```
// 默认情况下参数会按值传递
static int Add(int x, int y)
{
  int ans = x + y;

  // 由于我们修改的是原始数据的副本，调用者不会看到这些改变
  x = 10000;
  y = 88888;
  return ans;
}
```

数值数据属于值类型。因此，如果在成员的作用域内修改参数的值，改变的就是调用者数据值的副本，调用者完全不会意识到这种改变。

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****\n");

  // 按值传入两个变量
  int x = 9, y = 10;
  Console.WriteLine("Before call: X: {0}, Y: {1}", x, y);
  Console.WriteLine("Answer is: {0}", Add(x, y));
  Console.WriteLine("After call: X: {0}, Y: {1}", x, y);
  Console.ReadLine();
}
```

和我们期望的那样，x和y的值在调用Add()之前和之后保持不变，因为数据点是按值传递的，如下所示。Add()操作的是数据的副本，因而调用者不会意识到Add()方法参数的任何改变。

```
***** Fun with Methods *****

Before call: X: 9, Y: 10
Answer is: 19
After call: X: 9, Y: 10
```

4.1.2　out修饰符

接下来，我们将使用输出参数。定义为带有输出参数（通过关键字out）的方法有义务在退出这个方法之前，给参数赋一个恰当的值（如果不这样做，就会产生编译器错误）。

例如，下面是另一个版本的Add()方法，它使用C#的out修饰符返回两个整数的和（注意，这个方法实际的返回值现在是void）：

```
// 输出参数由被调用的方法赋值
static void Add(int x, int y, out int ans)
{
  ans = x + y;
}
```

调用一个带有输出参数的方法也需要使用out修饰符。但是作为输出变量传递的本地变量在将它们作为输出变量传递前不需要赋值（如果这样做了，原来的值在调用后就会丢失），编译器允许你传递未分配的数据，原因在于所调用的方法内部必须包含这种分配。例如：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
...
  // 不需要给作为输出参数的本地变量赋初值，假设第一次使用它们时是将其作为输出参数的
  int ans;
  Add(90, 90, out ans);
  Console.WriteLine("90 + 90 = {0}", ans);
  Console.ReadLine();
}
```

前面的例子意在说明，实际上没有必要用输出参数返回这两个数的和。然而，C#的out修饰符的确有一个很有用的用途：通过它，调用者只使用一次方法调用就能获得多个返回值。

```
// 返回多个输出参数
static void FillTheseValues(out int a, out string b, out bool c)
{
  a = 9;
  b = "Enjoy your string.";
  c = true;
}
```

调用者可以调用FillTheseValues()方法。请注意，在调用和实现该方法时，必须使用out修饰符。

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
...
  int i; string str; bool b;
  FillTheseValues(out i, out str, out b);

  Console.WriteLine("Int is: {0}", i);
  Console.WriteLine("String is: {0}", str);
  Console.WriteLine("Boolean is: {0}", b);
  Console.ReadLine();
}
```

最后，如果方法定义了输出参数，就必须在退出方法之前为这个参数赋一个有效值。因此，由于输出参数没有在方法域中被赋值，如下方法会导致编译器错误：

```
static void ThisWontCompile(out int a)
{
  Console.WriteLine("Error! Forgot to assign output arg!");
}
```

4.1.3 ref修饰符

现在考虑C#的ref参数修饰符的用法。如果希望方法可以对在调用者作用域中声明的不同数据进行操作（通常是改变它的值），例如，排序和交换例程，就必须使用引用参数。注意输出参数和引用参数之间的区别。

- 输出参数不需要在它们被传递给方法之前初始化，因为方法在退出之前必须为输出参数赋值。
- 引用参数必须在它们被传递给方法之前初始化，因为是在传递一个对已存在变量的引用。如果不赋给它初始值，就相当于要对一个未赋值的本地变量进行操作。

下面通过交换两个string变量的方式(此处可使用任意两个其他的数据类型，包括int、bool、float等）来看看ref关键字的用法：

```
// 引用参数
public static void SwapStrings(ref string s1, ref string s2)
{
  string tempStr = s1;
  s1 = s2;
  s2 = tempStr;
}
```

这个方法可以这样被调用：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
...
  string str1 = "Flip";
  string str2 = "Flop";
  Console.WriteLine("Before: {0}, {1} ", str1, str2);
  SwapStrings(ref str1, ref str2);
  Console.WriteLine("After: {0}, {1} ", str1, str2);
  Console.ReadLine();
}
```

这里，调用者给局部字符串数据（str1和str2）赋了初始值。一旦SwapStrings()的调用返回，str1现在包含的值就为"Flop"，而str2的值为"Flip"，如下所示：

```
Before: Flip, Flop
After: Flop, Flip
```

说明 4.5节会再次讨论C#的ref关键字。我们可以看到，这个关键字的行为会根据参数是值类型（结构）还是引用类型（类）而变化。

4.1.4 params修饰符

C#使用params关键字支持参数数组的使用。要理解这个语言特性，就必须理解如何操作C#数组。如果不是这样的话，你可能会希望在读完4.2节之后再回到这里。

params关键字可以把可变数量的参数（相同类型）作为单个逻辑参数传给方法。同样，如果调用者传入强类型数组或以逗号分隔的项列表，以params关键字标记的参数就可以被处理。恩，的确有些乱。为了理清思路，让我们创建一个函数，它允许调用者传入任何数量的参数并返回计算后的平均值。

如果我们采用这个方法作为原型来接受double数组，就需要强制调用者首先定义数组，然后填充数组，最后把它传给方法。然而，如果我们定义了CalculateAverage()来接收double[]数据类型的params，调用者只需要传入以逗号分隔的double列表。.NET运行库会在后台自动把这组double包装成为一个double类型的数组。

```
// 返回一些double型的平均值
static double CalculateAverage(params double[] values)
{
  Console.WriteLine("You sent me {0} doubles.", values.Length);

  double sum = 0;
  if(values.Length == 0)
    return sum;

  for (int i = 0; i < values.Length; i++)
    sum += values[i];
  return (sum / values.Length);
}
```

这个方法定义为带有一个double型参数数组。这个方法相当于“在说”：“传递给我任意个double型（包括0）的数，我会计算它们的平均值。”因此，可以用下列任何一种方式调用CalculateAverage()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
...
  // 传递进一个以逗号分隔的double型数的列表
  double average;
  average = CalculateAverage(4.0, 3.2, 5.7, 64.22, 87.2);
  Console.WriteLine("Average of data is: {0}", average);

  // …或传递一个double型的数组
  double[] data = { 4.0, 3.2, 5.7 };
  average = CalculateAverage(data);
  Console.WriteLine("Average of data is: {0}", average);

  // 0的平均值是0
  Console.WriteLine("Average of data is: {0}", CalculateAverage());
  Console.ReadLine();
}
```

如果在定义CalculateAverage()时没有使用params修饰符，在第一次调用该方法时将导致编译器错误，因为编译器需要的是包含5个double参数的CalculateAverage()。

说明　为了避免歧义，C#要求方法只支持一个params参数，而且必须是参数列表中的最后一个参数。

你可能也猜到了，这项技术只不过对调用者来说方便了一些，因为CLR还是需要创建数组。如果数组在被调用的方法域中，我们完全可以把它当做一个完整的.NET数组，其中包含了System.Array基础类库类型的所有功能。考虑如下输出：

```
You sent me 5 doubles.
Average of data is: 32.864
You sent me 3 doubles.
Average of data is: 4.3
You sent me 0 doubles.
Average of data is: 0
```


4.1.5 定义可选参数

C#程序员还可以创建包含可选参数的方法。这项技术允许方法的调用者不指定不必要的参数，而是使用这些参数的默认值。

说明 如第16章所述，为C#加入可选参数的一个主要原因是简化与COM对象的互操作。一些微软的对象模型（如Office）的功能是通过COM对象公开的，而这些很久之前编写的代码使用了可选参数。

为了演示可选参数，假设我们有一个定义了可选参数的EnterLogData()方法：

```
static void EnterLogData(string message, string owner = "Programmer")
{
  Console.Beep();
  Console.WriteLine("Error: {0}", message);
  Console.WriteLine("Owner of Error: {0}", owner);
}
```

这里的最后一个string参数使用参数定义中的赋值语句设置了默认值"Programmer"。这样，我们就可以通过两种方式在Main()中调用EnterLogData()了：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
...
  EnterLogData("Oh no! Grid can't find data");
  EnterLogData("Oh no! I can't find the payroll data", "CFO");

  Console.ReadLine();
}
```

由于对EnterLogData()方法的第一次调用并没有指定第二个string参数，我们就知道应该由程序员对丢失数据负责，而CFO没有得到薪酬数据（因为在第二次方法调用的时候指定了第二个参数）。

非常重要的一点是，分配给可选参数的值必须在编译时确定，而不能在运行时确定（否则将得到编译时错误）。假设我们修改了EnterLogData()，增加了一个可选参数，如下所示：

```
// 错误！可选参数的默认值必须在编译时确定
static void EnterLogData(string message,
            string owner = "Programmer", DateTime timeStamp = DateTime.Now)
```

```
{
  Console.Beep();
  Console.WriteLine("Error: {0}", message);
  Console.WriteLine("Owner of Error: {0}", owner);
  Console.WriteLine("Time of Error: {0}", timeStamp);
}
```

这将无法通过编译，因为DataTime类的Now属性是在运行时而不是在编译时处理的。

说明　为了避免歧义，可选参数必须放在方法签名的最后。将可选参数放到非可选参数的前面将导致编译时错误。

4.1.6　使用命名参数调用方法

另一个C#语言特性是支持命名参数。说实话，乍看上去这个语言结构除了带来混淆的代码外，似乎没什么作用。再说一句实话，可能真的是这样！与可选参数一样，支持命名参数的主要原因也是为了简化与COM的互操作（参见第16章）。

命名参数允许你在调用方法时以任意顺序指定参数的值。因此，你可以使用冒号操作符通过名称来指定参数，而不必按位置传递参数（就像你在大多数情况下所做的那样）。为了演示命名参数的用法，假设我们的Program类中包含如下的方法：

```
static void DisplayFancyMessage(ConsoleColor textColor,
  ConsoleColor backgroundColor, string message)
{
  // 在消息打印前保存旧的颜色
  ConsoleColor oldTextColor = Console.ForegroundColor;
  ConsoleColor oldbackgroundColor = Console.BackgroundColor;

  // 设置新的颜色并打印消息
  Console.ForegroundColor = textColor;
  Console.BackgroundColor = backgroundColor;

  Console.WriteLine(message);

  // 恢复原来的颜色
  Console.ForegroundColor = oldTextColor;
  Console.BackgroundColor = oldbackgroundColor;
}
```

按现在DisplayFancyMessage()方法的书写方式，你可能希望调用者在调用该方法时先传递两个ConsoleColor变量，然后是一个string类型。不过，像下面这样使用命名参数进行调用也完全没有问题：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Methods *****");
...
  DisplayFancyMessage(message: "Wow!  Very Fancy indeed!",
    textColor: ConsoleColor.DarkRed,
    backgroundColor: ConsoleColor.White);

  DisplayFancyMessage(backgroundColor: ConsoleColor.Green,
    message: "Testing...",
```

```
    textColor: ConsoleColor.DarkBlue);
  Console.ReadLine();
}
```

关于命名参数的一个小问题是，如果你在调用方法的时候使用了位置参数，那么它们必须列在所有命名参数之前。换句话说，命名参数必须放置在方法调用的最后。例如，下面的代码：

```
// 没有问题，因为位置参数在命名参数之前
DisplayFancyMessage(ConsoleColor.Blue,
  message: "Testing...",
  backgroundColor: ConsoleColor.White);

// 错误，因为位置参数在命名参数之后
DisplayFancyMessage(message: "Testing...",
  backgroundColor: ConsoleColor.White,
  ConsoleColor.Blue);
```

除了这项限制，你可能仍然不清楚究竟在什么时候使用这个语言特性。毕竟，如果你要为一个方法指定3个参数，何苦费力不讨好地变换它们的顺序呢？

然而事实证明，如果你定义了一个包含可选参数的方法，命名参数就非常有用了。假设DisplayFancyMessage()现在支持可选参数，并且设置了合适的默认值：

```
static void DisplayFancyMessage(ConsoleColor textColor = ConsoleColor.Blue,
  ConsoleColor backgroundColor = ConsoleColor.White,
  string message = "Test Message")
{
   ...
}
```

由于每个参数都包含默认值，调用者就可以使用命名参数只指定那些不希望用默认值的参数。因此，如果调用者希望在白色背景上显示蓝色文本“Hello”，只需要这样：

```
DisplayFancyMessage(message: "Hello!");
```

或者，如果调用者希望在绿色背景上显示蓝色文本“Test Message”，可以这样调用DisplayFancyMessage()：

```
DisplayFancyMessage(backgroundColor: ConsoleColor.Green);
```

如你所见，可选参数和命名参数往往会一起使用。在研究构建C#方法的最后部分，我要介绍的话题是方法重载。

源代码 FunWithMethods应用程序位于Chapter 4子目录下。

4.1.7 成员重载

和其他现代的面向对象语言一样，C#允许方法重载。简而言之，当我们定义一组名字相同的成员时，如果它们的参数数量（或类型）不同，这样的成员就叫做被重载。

为了理解为什么重载这么有用，让我们从VB6开发者的角度来考虑一下。假设我们使用VB6构建了一组返回各种传入类型（Integer、Double等）的求和的方法。由于VB6不支持方法重载，我们就只能定义唯一一组本质上做同一件事情（返回参数的总和）的方法：

```
' VB6代码示例
Public Function AddInts(ByVal x As Integer, ByVal y As Integer) As Integer
  AddInts = x + y
End Function

Public Function AddDoubles(ByVal x As Double, ByVal y As Double) As Double
  AddDoubles = x + y
End Function

Public Function AddLongs(ByVal x As Long, ByVal y As Long) As Long
  AddLongs = x + y
End Function
```

这样的代码不仅不易于维护，而且调用者必须痛苦地记住每一个方法的名字。使用重载，就可以允许调用者调用一个叫做Add()的方法。同样，关键是要确保方法的每一个版本都有不同的参数组（只是返回类型不同的成员不够唯一）。

说明　在第9章中我们会解释，我们可以构建泛型方法，以进一步提升重载的概念。使用泛型，我们可以为方法的实现定义类型占位符，并且在调用成员时进行指定。

让我们新建一个控制台应用程序MethodOverloading来亲自体验一下。现在考虑如下的类定义：

```
// C#代码
class Program
{
  static void Main(string[] args)
  {
  }

  // 重载的Add()方法
  static int Add(int x, int y)
  { return x + y; }

  static double Add(double x, double y)
  { return x + y; }

  static long Add(long x, long y)
  { return x + y; }
}
```

现在，调用者只需要使用必需的参数调用Add()方法，因为编译器可以通过所提供的参数来解析合适的实现，从而进行调用，所以这样的语法完全可以通过编译：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Method Overloading *****\n");

  // 调用int版本的Add()
  Console.WriteLine(Add(10, 10));

  // 调用long版本的Add()
  Console.WriteLine(Add(900000000000, 900000000000));

  // 调用double版本的Add()
  Console.WriteLine(Add(4.3, 4.4));
```

```
    Console.ReadLine();
  }
```

在调用重载方法的时候，Visual Studio IDE提供了协助。当输入重载方法的名字时（例如我们熟悉的Console.WriteLine()），智能感知会列出这个方法所有版本的列表。可以在图4-1中看到，我们可以通过使用键盘的上下键在各个重载方法之间切换。

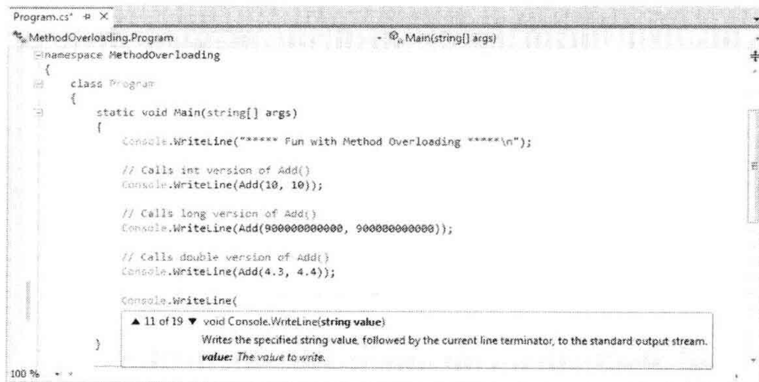


图4-1　Visual Studio对重载成员的智能感知

源代码　MethodOverloading项目的源代码位于Chapter 4子目录下。

这样，有关使用C#语法构建方法的基本研究到此结束。接着，让我们来看看如何构建和操作数组、枚举以及结构。

4.2　C#数组

你应该已经知道，数组是一组通过数字索引来访问的数据项。更精确地说，数组是一组相同类型的数据点（int数组、string数组、SportsCar数组等）。使用C#声明数组很简单。让我们新建一个控制台应用程序项目FunWithArrays来演示，在项目中有一个会从Main()调用的叫做SimpleArrays()的辅助方法：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Arrays *****");
    SimpleArrays();
    Console.ReadLine();
  }

  static void SimpleArrays()
  {
    Console.WriteLine("=> Simple Array Creation.");
    // 赋值一个包含3个元素的整数数组，编号为0、1、2
    int[] myInts = new int[3];

    // 初始化一个100项的字符串数组，编号0～99
```

```
    string[] booksOnDotNet = new string[100];
    Console.WriteLine();
  }
}
```

仔细看前面那段代码的注释。如果使用这个语法声明C#数组的话，数组声明中的数字就表示项的总数，而不是上界。还应注意，数组的下界总是从0开始，因此，如果我们写int[] myInts = new int[3]，最后我们会得到一个包含3个元素的数组（编号0~2）。

定义了数组变量后，就可以使用索引来填充元素索引了。更新后的SimpleArrays()方法如下所示：

```
static void SimpleArrays()
{
  Console.WriteLine("=> Simple Array Creation.");
  // 创建数组并且填充3个整数
  int[] myInts = new int[3];
  myInts[0] = 100;
  myInts[1] = 200;
  myInts[2] = 300;

  // 现在输出每一个值
  foreach(int i in myInts)
    Console.WriteLine(i);
  Console.WriteLine();
}
```

说明　如果我们声明数组，而不是显式填充每个索引，那么，每一个项都会被设置为数据类型的默认值（例如，bool的数组就被设置为false，int的数组就被设置为0，以此类推）。

4.2.1　C#数组初始化语法

除了逐个元素填充数组之外，还可以使用C#数组初始化语法来填充数组的元素，通过在花括号（{}）内指定每一个数组项来实现。如果我们需要创建一个已知大小的数组，并且希望快速指定初始值，这个语法就很有用。例如，下面是另一种方式的数组声明：

```
static void ArrayInitialization()
{
  Console.WriteLine("=> Array Initialization.");

  // 使用new关键字的数组初始化语法
  string[] stringArray = new string[]
    { "one", "two", "three" };
  Console.WriteLine("stringArray has {0} elements", stringArray.Length);

  // 不使用new关键字的数组初始化语法
  bool[] boolArray = { false, false, true };
  Console.WriteLine("boolArray has {0} elements", boolArray.Length);

  // 使用new关键字和大小的数组初始化
  int[] intArray = new int[4] { 20, 22, 23, 0 };
  Console.WriteLine("intArray has {0} elements", intArray.Length);
  Console.WriteLine();
}
```

注意，当使用“花括号”语法时，不需要指定数组大小（如构建stringArray类型的变量时），因为这

可以通过花括号中项的个数进行推断。还要注意，new关键字是可选的（如构建boolArray类型的变量时）。

在intArray声明的例子中，要记住指定的数字值是数组中元素的总数而不是上界值。如果声明的大小和初始化的个数不匹配，就会收到一个编译器错误，如下所示：

```
// 哦！大小和元素不匹配
int[] intArray = new int[2] { 20, 22, 23, 0 };
```

4.2.2 隐式类型本地数组

4

我们在上一章中学习了隐式类型本地变量，它由var关键字定义，其实际类型由编译器确定。同样，var关键字也可以用来定义隐式类型本地数组。这样我们在分配新数组变量的时候，就不需要指定数组本身所包含的类型。

```
static void DeclareImplicitArrays()
{
  Console.WriteLine("=> Implicit Array Initialization.");

  // a实际上是int[]
  var a = new[] { 1, 10, 100, 1000 };
  Console.WriteLine("a is a: {0}", a.ToString());

  // b实际上是double[]
  var b = new[] { 1, 1.5, 2, 2.5 };
  Console.WriteLine("b is a: {0}", b.ToString());

  // c实际上是string[]
  var c = new[] { "hello", null, "world" };
  Console.WriteLine("c is a: {0}", c.ToString());
  Console.WriteLine();
}
```

当然，在你用C#的隐式语法分配数组的时候，数组的初始化列表中每一项的类型都应该是相同的（例如，全都是int、string或SportsCar）。与你猜的不太一样，隐式类型本地数组的项默认不是System.Object，因此下面的代码将生成编译时错误：

```
// 错误！混合类型
var d = new[] { 1, "one", 2, "two", false };
```

4.2.3 定义object数组

在大部分情况下，在定义数组的时候可以指定保存在数组变量中的项类型。这看起来很简单，但有一点需要注意。在第6章中，我们会知道，System.Object是.NET类型系统中所有类型（包括基本数据类型）的最终基类。基于这一点，如果定义了一个System.Object的数组，子项就可以是任何东西。考虑如下ArrayOfObjects()方法（同样，可以从Main()调用进行测试）：

```
static void ArrayOfObjects()
{
  Console.WriteLine("=> Array of Objects.");

  // 对象数组可以是任何东西
  object[] myObjects = new object[4];
  myObjects[0] = 10;
```

```
    myObjects[1] = false;
    myObjects[2] = new DateTime(1969, 3, 24);
    myObjects[3] = "Form & Void";
    foreach (object obj in myObjects)
    {
      // 输出数组中每一项的类型和值
      Console.WriteLine("Type: {0}, Value: {1}", obj.GetType(), obj);
    }
    Console.WriteLine();
  }
```

在这里，我们遍历myObjects的内容，并使用System.Object的GetType()方法输出每一项的实际类型以及当前项的值。在这里我们不会过多研究有关System.Object.GetType()方法的细节，只需要理解这个方法用于获取项的完全限定名（第15章会完整研究类型信息和反射服务相关主题）。下面的输出为调用ArrayOfObjects()的结果。

```
=> Array of Objects.
Type: System.Int32, Value: 10
Type: System.Boolean, Value: False
Type: System.DateTime, Value: 3/24/1969 12:00:00 AM
Type: System.String, Value: Form & Void
```

4.2.4 使用多维数组

除了前面我们已经见到的一维数组以外，C#还支持两种多维数组。第一个叫做矩形数组，它只是一个每一行长度都相同的多维数组。如下代码声明并填充一个多维矩形数组：

```
static void RectMultidimensionalArray()
{
  Console.WriteLine("=> Rectangular multidimensional array.");
  // 矩形多维数组
  int[,] myMatrix;
  myMatrix = new int[6,6];

  // 填充6*6数组
  for(int i = 0; i < 6; i++)
    for(int j = 0; j < 6; j++)
      myMatrix[i, j] = i * j;

  // 输出6*6数组
  for(int i = 0; i < 6; i++)
  {
    for(int j = 0; j < 6; j++)
      Console.Write(myMatrix[i, j] + "\t");
    Console.WriteLine();
  }
  Console.WriteLine();
}
```

第二种多维数组的类型叫做交错数组。顾名思义，交错数组包含一些内部数组，每一个都有各自的上界，例如：

```
static void JaggedMultidimensionalArray()
{
```

```
    Console.WriteLine("=> Jagged multidimensional array.");
    // 交错多维数组 （也就是数组的数组）
    // 声明一个具有5个不同数组的数组
    int[][] myJagArray = new int[5][];

    // 创建交错数组
    for (int i = 0; i < myJagArray.Length; i++)
      myJagArray[i] = new int[i + 7];

    // 输出每一行（记住，每一个元素都默认为0）
    for(int i = 0; i < 5; i++)
    {
      for(int j = 0; j < myJagArray[i].Length; j++)
        Console.Write(myJagArray[i][j] + " ");
      Console.WriteLine();
    }
    Console.WriteLine();
  }
```


图4-2展示了调用Main()方法中的RectMultidimensionalArray()和JaggedMultidimensionalArray()方法的输出。

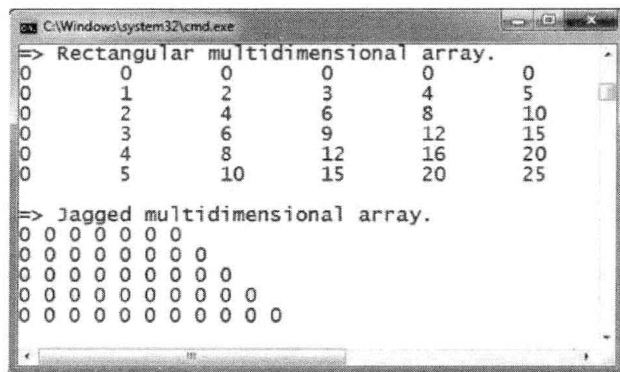


图4-2　矩形多维数组和交错多维数组

4.2.5　数组作为参数（和返回值）

只要我们创建了一个数组，就完全可以把它作为参数进行传递或作为成员返回值接收。例如，如下PrintArray()方法接收传入的int数组并将每一个成员输出到控制台，而GetStringArray()则填充string数组并返回给调用者：

```
static void PrintArray(int[] myInts)
{
  for(int i = 0; i < myInts.Length; i++)
    Console.WriteLine("Item {0} is {1}", i, myInts[i]);
}

static string[] GetStringArray()
{
  string[] theStrings = {"Hello", "from", "GetStringArray"};
  return theStrings;
}
```

这些方法的调用方式我们也能想到：

```
static void PassAndReceiveArrays()
{
  Console.WriteLine("=> Arrays as params and return values.");
  // 传递数组作为参数
  int[] ages = {20, 22, 23, 0} ;
  PrintArray(ages);

  // 获取数组作为返回值
  string[] strs = GetStringArray();
  foreach(string s in strs)
    Console.WriteLine(s);

  Console.WriteLine();
}
```

那么，至此你应该能很好地理解定义、填充和获取C#数组变量内容的整个过程了。为了完善这个知识体系，现在来研究一下System.Array类的作用。

4.2.6 System.Array基类

我们创建的每一个数组都从System.Array类获得了很多功能。使用这些公共成员，我们就能使用统一的对象模型来操作数组。表4-2列出其中一些有趣的成员（对于完整细节，请参阅.NET Framework 4.5 SDK）。

表4-2　System.Array的部分成员

Array类的成员	作　用
Clear()	这个静态方法将数组中一系列元素设置为空值（值项为0，对象引用为null，布尔值为false）
CopyTo()	这个方法用来将源数组中的元素复制到目标数组中
Length	这个属性返回数组中项的个数
Rank	这个属性返回当前数组维数
Reverse()	这个静态方法反转一维数组的内容
Sort()	这个静态方法为内建类型的一维数组排序。如果数组中的元素实现了IComparer接口，我们就可以为自定义类型排序（见第9章）

让我们实际运用一下这些成员。下面的辅助方法使用了Reverse()和Clear()方法来提取有关数组的信息并输出到控制台。

```
static void SystemArrayFunctionality()
{
  Console.WriteLine("=> Working with System.Array.");
  // 初始化起始项
 string[] gothicBands = {"Tones on Tail", "Bauhaus", "Sisters of Mercy"};

  // 按声明的次序输出名字
  Console.WriteLine("-> Here is the array:");
  for (int i = 0; i < gothicBands.Length; i++)
  {
    // 输出一个名字
```

```
    Console.Write(gothicBands[i] + ", ");
  }
  Console.WriteLine("\n");

  // 反转它们
  Array.Reverse(gothicBands);
  Console.WriteLine("-> The reversed array");
  // 输出它们
  for (int i = 0; i < gothicBands.Length; i++)
  {
    // 输出一个名字
    Console.Write(gothicBands[i] + ", ");
  }
  Console.WriteLine("\n");

  // 清除除了最后成员之外的所有项
  Console.WriteLine("-> Cleared out all but one...");
  Array.Clear(gothicBands, 1, 2);
  for (int i = 0; i < gothicBands.Length; i++)
  {
    // 输出一个名字
    Console.Write(gothicBands[i] + ", ");
  }
  Console.WriteLine();
}
```

如果从Main()中调用这个方法，会得到如下所示的输出结果。

```
=> Working with System.Array.
-> Here is the array:
Tones on Tail, Bauhaus, Sisters of Mercy,

-> The reversed array
Sisters of Mercy, Bauhaus, Tones on Tail,

-> Cleared out all but one...
Sisters of Mercy, , ,
```

注意，System.Array的很多成员都定义为静态方法，因此可以在类级别进行调用（如Array.Sort()或Array.Reverse()方法）。这样的方法都需要传入到我们希望处理的数组。System.Array的其他方法（Length属性）绑定在对象级别上，因此我们可以直接在数组上调用成员。

源代码 FunWithArrays项目的源代码位于Chapter 4子目录下。

4.3 枚举类型

第1章介绍过，.NET类型系统由类、结构、枚举、接口和委托组成。让我们从枚举（enumeration，也可以简写为enum）开始对这些类型的考查，首先新建一个名为FunWithEnums的控制台应用程序。

注意　不要混淆“枚举”（enum）和“枚举器”（enumerator），它们是完全不同的概念。枚举是自定义的名/值对的数据类型。枚举器是实现了.NET接口IEnumerable的类或结构。通常来说，集合类和System.Array类会实现该接口。你将在第8章看到，支持IEnumerable的对象可以用于foreach循环。

在构建系统的时候，创建一组符号名来对应已知的数字值会很方便。例如，如果创建一个工资系统，我们可能会希望使用诸如副总裁、经理、职员、实习生等常量来指代员工类型。C#支持自定义枚举的概念来满足这种需求。例如，下面是一个名为EmpType的枚举：

```
// 自定义枚举
enum EmpType
{
  Manager,       // = 0
  Grunt,         // = 1
  Contractor,    // = 2
  VicePresident  // = 3
}
```

EmpType枚举定义了4个命名常量来对应一些离散的数值。默认情况下，第一个元素被设置为值0，其余的按照n+1递推。如果需要的话，我们完全可以改变初始值。例如，如果需要EmpType的成员是102~105，我们可以这样做：

```
// 从102开始
enum EmpType
{
  Manager = 102,
  Grunt,        // = 103
  Contractor,   // = 104
  VicePresident // = 105
}
```

枚举不一定是连续的，也不需要有唯一值。如果（由于种种原因）按如下所示创建EmpType是有意义的话，当然也可以通过编译：

```
// 枚举的元素不需要是连续的
enum EmpType
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

4.3.1　控制枚举的底层存储

默认情况下，用来保存枚举值的存储类型是System.Int32（C# int），当然也可以改成我们喜欢的类型。C#枚举可以以相似的方式定义为核心系统类型（byte、short、int或long）。例如，如果希望将EmpType的实际存储值设置为一个byte而不是一个int，可以这么写：

```
// 这次，EmpType对应实际的byte
enum EmpType : byte
{
```

```
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 9
}
```

如果我们构建的.NET应用程序将会被部署在低内存的设备中（如支持.NET的手机或PDA），并且希望尽可能节省内存，那么改变枚举的实际类型可能会很有用。当然，如果确实使用byte作为存储来创建枚举，每一个值就必须在其范围之内！例如，下面的EmpType会导致编译器错误，因为值999所占的空间大于一个字节：

```
// 编译器错误！999所占的空间大于一个字节
enum EmpType : byte
{
  Manager = 10,
  Grunt = 1,
  Contractor = 100,
  VicePresident = 999
}
```


4.3.2 声明枚举变量

设定了枚举的范围和存储类型之后，就可以使用它来替代所谓的幻数了。因为枚举只不过是用户自定义的类型，我们可以把它们作为函数的返回值、方法参数、本地变量等。假设我们有一个叫做AskForBonus()的方法，它接受EmpType变量作为唯一的参数。根据传入参数的值，我们输出对应奖金请求的响应：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("**** Fun with Enums *****");
    // 创建职员的类型
    EmpType emp = EmpType.Contractor;
    AskForBonus(emp);
    Console.ReadLine();
  }

  // 用枚举作为参数
  static void AskForBonus(EmpType e)
  {
    switch (e)
    {
      case EmpType.Manager:
        Console.WriteLine("How about stock options instead?");
      break;
      case EmpType.Grunt:
        Console.WriteLine("You have got to be kidding...");
      break;
      case EmpType.Contractor:
        Console.WriteLine("You already get enough cash...");
      break;
      case EmpType.VicePresident:
        Console.WriteLine("VERY GOOD, Sir!");
```

```
        break;
      }
    }
  }
```

注意，为枚举变量赋值时，必须以枚举名（EmpType）来设置值（Grunt）。因为枚举是固定的一组名称/值对，将枚举变量设置为枚举类型没有定义的值是不合法的：

```
static void ThisMethodWillNotCompile()
{
  // 错误！SalesManager不在EmpType枚举中
  EmpType emp = EmpType.SalesManager;

  // 错误！忘记设定Grunt值为EmpType枚举
  emp = Grunt;
}
```

4.3.3　System.Enum类型

.NET枚举从System.Enum类类型获得了很多功能。这个类定义了许多用来查询和转换某个枚举的方法。一个很有用的方法就是静态的Enum.GetUnderlyingType()方法，顾名思义，它返回用于保存枚举类型值的数据类型（对于当前的EmpType声明，就是System.Byte）。

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  // 创建职员的类型
  EmpType emp = EmpType.Contractor;
  AskForBonus(emp);

  // 输出枚举的存储
  Console.WriteLine("EmpType uses a {0} for storage",
    Enum.GetUnderlyingType(emp.GetType()));
  Console.ReadLine();
}
```

如果我们查阅Visual Studio对象浏览器，就会发现Enum.GetUnderlyingType()方法需要我们传入System.Type作为第一个参数。Type表示某个.NET实体的元数据描述，这在第15章中会详细介绍。

获取元数据的一个可行方式（前面提到过）是使用GetType()方法，这个方法是所有.NET基础类库中的类型所共有的。另外一种方式是使用C#的typeof操作符。这样做的好处是，不需要我们持有希望获取元数据描述的实体的变量。

```
// 使用typeof获取一个Type
Console.WriteLine("EmpType uses a {0} for storage",
    Enum.GetUnderlyingType(typeof(EmpType)));
```

4.3.4　动态获取枚举的名称/值对

除了Enum.GetUnderlyingType()方法之外，所有C#枚举都支持ToString()方法，它返回当前枚举值的字符串名。例如：

```
static void Main(string[] args)
{
```

```
  Console.WriteLine("**** Fun with Enums *****");
  EmpType emp = EmpType.Contractor;
  ASKForBonus(emp);

  // 输出“emp is a Contractor”
  Console.WriteLine("emp is a {0}.", emp.ToString());
  Console.ReadLine();
}
```

如果希望获取某个枚举变量值（而不是它的名字），只需要根据底层存储类型对枚举变量进行强制类型转换即可。例如：

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
  EmpType emp = EmpType.Contractor;
  ...

  // 输出“Contractor = 100”
  Console.WriteLine("{0} = {1}", emp.ToString(), (byte)emp);
  Console.ReadLine();
}
```

说明 静态的Enum.Format()方法通过指定期望的格式化标志来提供更好的格式化选项。有关System.Enum.Format()方法的完整细节，请参考.NET Framework 4.5 SDK文档。

System.Enum还定义了另外一个名为GetValues()的静态方法。这个方法返回System.Array的一个实例。数组中每一项都对应指定枚举的一个成员。考虑如下方法，它会输出作为参数传入的任何枚举中的每一个名称/值对：

```
// 这个方法会输出任何枚举的细节
static void EvaluateEnum(System.Enum e)
{
  Console.WriteLine("=> Information about {0}", e.GetType().Name);

  Console.WriteLine("Underlying storage type: {0}",
    Enum.GetUnderlyingType(e.GetType()));
  // 获取传入参数的名称/值对
  Array enumData = Enum.GetValues(e.GetType());
  Console.WriteLine("This enum has {0} members.", enumData.Length);
  // 现在使用D格式标志（见第3章）显示字符串名和关联的值
  for(int i = 0; i < enumData.Length; i++)
  {
    Console.WriteLine("Name: {0}, Value: {0:D}",
      enumData.GetValue(i));
  }
  Console.WriteLine();
}
```

为了测试这个新方法，我们更新Main()方法来创建几个在System命名空间中声明的枚举类型（EmpType枚举也可以用来测试）。例如：

```
static void Main(string[] args)
{
  Console.WriteLine("**** Fun with Enums *****");
```

```
    ...
    EmpType e2 = EmpType.Contractor;

    // 这些类型为System命名空间中的枚举值
    DayOfWeek day = DayOfWeek.Monday;
    ConsoleColor cc = ConsoleColor.Gray;

    EvaluateEnum(e2);
    EvaluateEnum(day);
    EvaluateEnum(cc);
    Console.ReadLine();
}
```

输出结果如图4-3所示。



图4-3 动态获取枚举类型的名称/值对

在本书中我们会看到，枚举在.NET基础类库中被广泛使用。例如，ADO.NET使用许多枚举来表示数据库连接的状态（已打开、已关闭等）、DataTable中行的状态（如被改变的、新的、分离的）等。因此，使用枚举时，可以通过System.Enum的成员来和名称/值对进行交互。

源代码 FunWithEnums项目的源代码位于Chapter 4子目录下。

4.4 结构类型

既然我们已经理解了枚举类型，接下来就研究一下.NET结构（structure，简称为struct）的使用。

结构类型很适合在应用程序中对数学、几何以及其他“原子”实体建模。结构（和枚举相似）是用户自定义的类型，然而，结构不只是一组名称/值对。结构是可以包含许多数据字段和操作这些字段的成员的类型。

说明　如果你有OOP背景，可以把结构看成是“轻量级的类类型”，因为结构提供了一种方式来定义这样一种类型，它们会支持封装，但不能用来构建一组相关类型。如果我们需要通过继承来构建一组相关类型，就需要使用类类型。

从表面上看，定义和使用结构的过程很简单，但是其中的一些细节很重要。现在开始对结构类型的研究，我们新建一个名为FunWithStructures的项目。在C#中，使用struct关键字来创建结构。定义一个新结构Point，它定义了两个int类型的成员变量和一组与上述数据交互的方法。

```
struct Point
{
  // 结构的字段
  public int X;
  public int Y;

  // 将(X,Y)坐标增加1
  public void Increment()
  {
    X++; Y++;
  }

  // 将(X,Y)坐标减去1
  public void Decrement()
  {
    X--; Y--;
  }

  // 显示当前坐标
  public void Display()
  {
    Console.WriteLine("X = {0}, Y = {1}", X, Y);
  }
}
```

这里使用public关键字定义了两个整型数据类型（X和Y），这个关键字是访问控制修饰符（第5章会详细介绍）。使用public关键字来声明数据可以确保调用者能直接获取某个Point变量的数据（通过点操作符）。

说明　通常在类或结构中定义公共数据是一个不好的方式。我们最好使用私有数据，它可以使用公共属性来访问和改变。第5章会详述这些细节。

Main()方法测试了一下我们的Point类型：

```
static void Main(string[] args)
{
  Console.WriteLine("***** A First Look at Structures *****\n");
```

```
    // 创建初始Point
    Point myPoint;
    myPoint.X = 349;
    myPoint.Y = 76;
    myPoint.Display();

    // 调整X和Y值
    myPoint.Increment();
    myPoint.Display();
    Console.ReadLine();
  }
```

输出结果和你期望的一样：

```
***** A First Look at Structures *****

X = 349, Y = 76
X = 350, Y = 77
```

创建结构变量

创建结构变量的方式有好几种。在这里，我们只是创建了一个Point变量，并在调用其成员之前为每一个公共字段数据赋值。如果我们在使用结构之前不为每一个公共字段数据（这里就是X和Y）赋值，就会收到一个编译器错误：

```
// 错误！没有为Y赋值
Point p1;
p1.X = 10;
p1.Display();

// 正确！在使用前两个字段都赋值了
Point p2;
p2.X = 10;
p2.Y = 10;
p2.Display();
```

另一种可行的方法是，使用C#的new关键字来创建结构变量，它会调用结构默认的构造函数。根据定义，默认的构造函数不接受任何输入参数。调用结构默认构造函数的好处是，每一个字段数据都会被自动设置为默认值：

```
// 使用默认构造函数将所有字段设置为默认值
Point p1 = new Point();

// 输出X=0,Y=0
p1.Display();
```

还可以使用自定义构造函数来设计结构。它允许我们在创建变量时指定字段数据的值，而不是逐个字段设置数据成员。第5章会详细研究构造函数，然而为了演示，让我们使用如下代码更新Point结构：

```
struct Point
{
  // 结构的字段
  public int X;
  public int Y;
```

```
  // 自定义的构造函数
  public Point(int XPos, int YPos)
  {
    X = XPos;
    Y = YPos;
  }
...
}
```

这样，我们就可以创建Point变量了：

```
// 调用自定义构造函数
Point p2 = new Point(50, 60);

// 输出X=50,Y=60
p2.Display();
```

之前提到过，结构的使用看上去很简单。但为了更好地理解这个类型，我们需要研究.NET值类型和.NET引用类型之间的区别。

源代码 FunWithStructures项目的源代码位于Chapter 4子目录下。

4.5 值类型和引用类型

说明 下面对值类型和引用类型的讨论的前提是，你已经有了面向对象编程的经验。如果没有面向对象编程的经验，你可能需要跳过本节和4.6节，在阅读第5章和第6章之后再来重新阅读这部分内容。

和数组、字符串或枚举不同，C#结构在.NET类库中没有完全同名的表示（也就是说，没有System.Structure类），但是它们都隐式派生自System.ValueType。简而言之，System.ValueType的作用是确保所有派生类型（如任何结构）都分配在栈上而不是垃圾回收堆上。创建和销毁分配在栈上的数据都很快，因为它的生命周期是由定义的作用域决定的。而分配在堆上的数据由.NET垃圾回收器监控，其生命周期的决定因素有很多，这将在第13章中介绍。

从功能上说，System.ValueType的唯一目的是，重载由System.Object定义的虚方法来使用基于值而不是基于引用的语法。你可能已经知道，重载会改变定义在基类中的虚（也可能是抽象的）方法的实现。ValueType的基类是System.Object。事实上，由System.ValueType定义的实例方法和由System.Object定义的完全一样：

```
// 结构和枚举隐式扩展了System.ValueType
public abstract class ValueType : object
{
  public virtual bool Equals(object obj);
  public virtual int GetHashCode();
  public Type GetType();
  public virtual string ToString();
}
```

由于值类型使用基于值的语法，结构（也包括所有数值数据类型int、float等，以及任何枚举或自定义结构）的生命周期是可以预测的。当结构变量离开定义域时，它就会立即从内存中移除：

```
// 本地变量在方法返回时弹出栈
static void LocalValueTypes()
{
  // "int"其实是System.Int32结构
  int i = 0;

  // Point是结构类型
  Point p = new Point();
} // "i"和"p"在这里弹出栈
```

4.5.1 值类型、引用类型和赋值操作符

当把一个值类型赋给另外一个时，就是对字段成员逐一进行复制。对于System.Int32这样的简单数据类型，唯一需要复制的成员就是数值。然而，对于我们的Point，X和Y值会被复制到新的结构变量中。例如，新建一个名为ValueAndReferenceTypes的控制台应用程序，并且将之前的Point定义复制到新的命名空间中。现在为Program类型增加如下的方法：

```
// 为两个内建的值类型赋值会在栈上产生两个独立变量
static void ValueTypeAssignment()
{
  Console.WriteLine("Assigning value types\n");

  Point p1 = new Point(10, 10);
  Point p2 = p1;

  // 输出两个Point
  p1.Display();
  p2.Display();

  // 改变p1.X并且输出。p2.X不会改变
  p1.X = 100;
  Console.WriteLine("\n=> Changed p1.X\n");
  p1.Display();
  p2.Display();
}
```

现在已经创建了一个Point类型的变量（命名为p1），并赋值给另外一个Point（p2）。由于Point是值类型，在栈上会有MyPoint类型的两个副本，每一个都可以被独立操作。因此，当改变p1.X的值时，p2.X不会受到影响：

```
Assigning value types

X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 10, Y = 10
```

和栈中的值类型相比，当对引用类型（也就是所有类实例）应用赋值操作符时，我们就是在内存

中重定向引用变量的指向。让我们新建一个类类型PointRef来说明，它和Point结构有几乎相同的成员，只不过重命名了构造函数来匹配类名。

```
// 类总是引用类型
class PointRef
{
  // 和Point结构有相同的成员

  // 确保将构造函数名改为PointRef
  public PointRef(int XPos, int YPos)
  {
    X = XPos;
    Y = YPos;
  }
}
```

现在，在如下新方法中使用PointRef类型。注意在其他地方使用PointRef类而不是Point结构，其代码和ValueTypeAssignment()方法完全一样。

```
static void ReferenceTypeAssignment()
{
  Console.WriteLine("Assigning reference types\n");
  PointRef p1 = new PointRef(10, 10);
  PointRef p2 = p1;

  // 输出两个Point ref
  p1.Display();
  p2.Display();

  // 改变p1.X并且再次输出
  p1.X = 100;
  Console.WriteLine("\n=> Changed p1.X\n");
  p1.Display();
  p2.Display();
}
```

在这里，有两个引用指向托管堆中的同一个对象。因此，当使用p1引用改变X值时，p2.X也报告了相同值。假定在Main()中调用了这个新方法，输出结果如下：

```
Assigning reference types

X = 10, Y = 10
X = 10, Y = 10
=> Changed p1.X
X = 100, Y = 10
X = 100, Y = 10
```

4.5.2 包含引用类型的值类型

现在你已经更好地理解了值类型和引用类型之间的区别，下面就分析一个更复杂的例子。假设有下面这个引用（类）类型，它保存着一个能够用自定义构造函数设置的信息字符串：

```
class ShapeInfo
{
  public string infoString;
```

```
    public ShapeInfo(string info)
    {
      infoString = info;
    }
  }
```

现在，假设要在名为Rectangle的值类型中包含这个类类型的变量。为了允许调用者设置内部的ShapeInfo成员变量的值，还提供了一个自定义的构造函数。下面是Rectangle类型的完整定义：

```
struct Rectangle
{
  // Rectangle结构包含一个引用类型成员
  public ShapeInfo rectInfo;

  public int rectTop, rectLeft, rectBottom, rectRight;

  public Rectangle(string info, int top, int left, int bottom, int right)
  {
    rectInfo = new ShapeInfo(info);
    rectTop = top;  rectBottom = bottom;
    rectLeft = left; rectRight = right;
  }

  public void Display()
  {
    Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}, " +
      "Left = {3}, Right = {4}",
      rectInfo.infoString, rectTop, rectBottom, rectLeft, rectRight);
  }
}
```

这里的值类型中包含了一个引用类型。一个非常重要的问题出现了：将一个Rectangle变量赋给另一个变量，会发生什么呢？由于我们已经了解了值类型，由此假设整型数据（事实上是个结构）对每一个Rectangle变量都应该是一个独立的实体，这是正确的。但内部的引用类型会怎样呢？是对象的状态将被完全复制，还是指向对象的引用被复制呢？为了回答这个问题，定义下面这个方法并且从Main()中调用它：

```
static void ValueTypeContainingRefType()
{
  // 创建第一个Rectangle
  Console.WriteLine("-> Creating r1");
  Rectangle r1 = new Rectangle("First Rect", 10, 10, 50, 50);

  // 现在将一个新的Rectangle赋值给r1
  Console.WriteLine("-> Assigning r2 to r1");
  Rectangle r2 = r1;

  // 改变r2的值
  Console.WriteLine("-> Changing values of r2");
  r2.rectInfo.infoString = "This is new info!";
  r2.rectBottom = 4444;

  // 输出两个rectangle的值
  r1.Display();
  r2.Display();
}
```

输出结果如下所示：

```
-> Creating r1
-> Assigning r2 to r1
-> Changing values of r2
String = This is new info!, Top = 10, Bottom = 50, Left = 10, Right = 50
String = This is new info!, Top = 10, Bottom = 4444, Left = 10, Right = 50
```

可以看到，当使用r2的引用改变信息字符串的值时，r1的引用显示了同样的值。默认情况下，当值类型包含其他引用类型时，赋值将生成一个引用的副本。这样就有两个独立的结构，每一个都包含指向内存中同一个对象的引用（也就是“浅复制”）。当想执行一个“深复制”，即将内部引用的状态完全复制到一个新对象中时，需要实现ICloneable接口（第8章会讲到）。

源代码 ValueAndReferenceTypes项目的源代码位于Chapter 4子目录下。

4.5.3 按值传递引用类型

引用类型或值类型显然可以作为参数传递给方法。但是，按引用传递一个引用类型（如类）与按值传递一个类型有很大的不同。为了理解这个区别，假设新的控制台应用程序RefTypeValType Params中有一个Person类，定义如下：

```
class Person
{
  public string personName;
  public int personAge;

  // 构造函数
  public Person(string name, int age)
  {
    personName = name;
    personAge = age;
  }
  public Person(){}

  public void Display()
  {
    Console.WriteLine("Name: {0}, Age: {1}", personName, personAge);
  }
}
```

现在，如果创建一个方法，允许调用者按值传入Person类型，会怎么样呢（注意，没有参数修饰符，如out或ref）？

```
static void SendAPersonByValue(Person p)
{
  // 改变"p"的年龄
  p.personAge = 99;

  // 调用者能看到这个重新赋值吗
  p = new Person("Nikki", 99);
}
```

注意SendAPersonByValue()方法是怎样试图将传入的Person引用重新赋给一个新对象，并且改变一

些状态数据的。现在使用下面的Main()方法来测试一下这个方法：

```
static void Main(string[] args)
{
  // 按值传递引用类型
  Console.WriteLine("***** Passing Person object by value *****");
  Person fred = new Person("Fred", 12);
  Console.WriteLine("\nBefore by value call, Person is:");
  fred.Display();

  SendAPersonByValue(fred);
  Console.WriteLine("\nAfter by value call, Person is:");
  fred.Display();
  Console.ReadLine();
}
```

这个调用的输出结果如下所示：

```
***** Passing Person object by value *****

Before by value call, Person is:
Name: Fred, Age: 12

After by value call, Person is:
Name: Fred, Age: 99
```

可以看出，personAge的值被修改了。这个行为看起来似乎违反了“按值”传递的语义。如果能够改变传入的Person的状态，那么复制的是什么？答案是：复制了指向调用者对象的引用。由于SendAPersonByValue()方法与调用者指向同一个对象，所以改变对象的状态数据是可能的。但是无法把引用重新赋值给一个新的对象[①]。

4.5.4 按引用传递引用类型

现在假设有一个SendAPersonByReference()方法，它按引用来传递引用类型（注意ref参数修饰符）：

```
static void SendAPersonByReference(ref Person p)
{
  // 改变"p"的一些数据
  p.personAge = 555;

  // "p"现在指向了堆上的一个新对象
  p = new Person("Nikki", 999);
}
```

以上代码使得被调用者操作传入参数时有完全的灵活性。被调用者不仅可以改变对象的状态，而且在改变对象状态后还可以将引用重新赋值为Person类型。考虑下面更新后的Main()方法：

```
static void Main(string[] args)
{
  // 按引用传递引用类型
  Console.WriteLine("***** Passing Person object by reference *****");
  ...
```

① 就像上面的p = new Person("Nikki", 99);，代码并没有起作用。有一点像C++中的常量指针。——译者注

```
    Person mel = new Person("Mel", 23);
    Console.WriteLine("Before by ref call, Person is:");
    mel.Display();

    SendAPersonByReference(ref mel);
    Console.WriteLine("After by ref call, Person is:");
    mel.Display();
    Console.ReadLine();
}
```

注意输出结果如下所示:

```
***** Passing Person object by reference *****
Before by ref call, Person is:
Name: Mel, Age: 23
After by ref call, Person is:
Name: Nikki, Age: 999
```

可以看到，在调用后对象mel返回一个名为Nikki的Person类型，这是因为方法可以改变传入的引用在内存中的指向。按引用传递引用类型时需要记住的黄金规则如下:

- 如果按引用传递引用类型，被调用者可能改变对象的状态数据的值和所引用的对象;
- 如果按值传递引用类型，被调用者可能改变对象的状态数据的值，但不能改变所引用的对象。

源代码 RefTypeValTypeParams项目的源代码位于Chapter 4子目录下。

4.5.5 值类型和引用类型：最后的细节

最后，表4-3总结了值类型和引用类型之间主要的区别。

表4-3 值类型和引用类型的比较

问　题	值　类　型	引用类型
这个类型分配在哪里	分配在栈上	分配在托管堆上
变量是怎样表示的	值类型变量是局部复制	引用类型变量指向被分配的实例所占的内存
基类型是什么	必须派生自System.ValueType	可以派生自除了System.ValueType以外的任何类型，只要那个类型不是密封的（关于密封的更多细节见第6章）
这个类型能作为其他类型的基类吗	不能。值类型总是密封的，不能被继承	能。如果这个类型不是密封的，它可以作为其他类型的基类
默认的参数传递行为是什么	变量是按值传递的（也就是说，一个变量的副本传入被调用的函数）	对于值类型，对象按值复制。对于引用类型，引用按值复制
这个类型能重写System.Object.Finalize()吗	不能。值类型不会放在堆上，因此不需要被终结	可以间接地重写（更多细节见第8章）
可以为这个类型定义构造函数吗	是的，但是默认的构造函数被保留（也就是自定义构造函数必须全部带有参数）	当然
这个类型的变量什么时候消亡	当它们越出定义的作用域时	当托管堆被垃圾回收时

尽管它们存在差异，但是值类型和引用类型都有实现接口的能力，并且可以支持任意数量的字段、方法、重载操作符、常量、属性和事件。

4.6　C#可空类型

在结束本章之前，让我们使用最后一个叫做NullableTypes的控制台应用程序来研究可空数据类型。我们知道，CLR数据类型有一个固定的范围，并且由System命名空间中的类型进行表示。例如，System.Boolean数据类型可以从集合{true,false}中赋值。现在我们应该记得所有数值数据类型（也包括Boolean数据类型）都是值类型。按照规则，null用来建立一个空的对象引用，所以值类型永远不可以被赋值为null。

```
static void Main(string[] args)
{
  // 编译器错误
  // 值类型不能设置为null
  bool myBool = null;
  int myInt = null;

  // 没错！字符串是引用类型
  string myString = null;
}
```

自从.NET 2.0发布之后，我们就可以创建可空数据类型了。简而言之，可空类型可以表示所有实际类型的值加上null。因此，如果声明一个可空的bool，就可以从集合{true, false, null}中进行赋值。如果和关系数据库打交道，这就会很有用，因为在数据库表中遇到未定义的列是很常见的事情。如果没有可空数据类型的概念，在C#中就没有很方便的方式来表示没有值的数值数据点。

为了定义一个可空变量类型，应在底层数据类型中添加问号（?）作为后缀。注意，这种语法只对值类型是合法的。如果试图创建一个可空引用类型（包括字符串），就会遇到编译时错误。与非可空变量一样，局部可空变量必须赋一个初始值才能使用。

```
static void LocalNullableVariables()
{
  // 定义一些局部可空类型
  int? nullableInt = 10;
  double? nullableDouble = 3.14;
  bool? nullableBool = null;
  char? nullableChar = 'a';
  int?[] arrayOfNullableInts = new int?[10];

  // 错误！字符串是引用类型
  // string? s = "oops";
}
```

在C#中，?后缀记法实际上是创建一个泛型System.Nullable<T>结构类型实例的简写。尽管第9章才分析泛型，但了解System.Nullable<T>类型提供了一组所有可空类型都可以使用的成员是很重要的。

例如，可以通过编程，用HasValue属性或!=操作符判断，一个可空变量是否确实被赋予了一个null值。可空类型被赋的值可以通过Value属性或直接获得。因为?后缀只是使用Nullable<T>的一种简化表示，所以可以按如下所示实现LocalNullableVariables()方法。

```
static void LocalNullableVariablesUsingNullable()
{
  // 使用Nullable<T>定义一些局部可空变量
  Nullable<int> nullableInt = 10;
  Nullable<double> nullableDouble = 3.14;
  Nullable<bool> nullableBool = null;
  Nullable<char> nullableChar = 'a';
  Nullable<int>[] arrayOfNullableInts = new int?[10];
}
```

4.6.1 使用可空类型

在涉及数据库编程时，可空数据类型可能特别有用，因为一个数据表中的列可能有意是空的（例如，未定义）。举个例子，假设有下面的类，模拟访问一个数据库的过程，该数据库包含一个表，其中有两个可能为null的列。注意，GetIntFromDatabase()方法没有给可空整型成员变量赋值，GetBoolFromDatabase()给bool?成员赋了一个合法的值：

```
class DatabaseReader
{
  // 可空数据字段
  public int? numericValue = null;
  public bool? boolValue = true;

  // 注意可空返回类型
  public int? GetIntFromDatabase()
  { return numericValue; }

  // 注意可空返回类型
  public bool? GetBoolFromDatabase()
  { return boolValue; }
}
```

现在，假设有下面的Main()方法，调用了DatabaseReader类的每一个成员，并使用HasValue和Value成员以及C#相等操作符（准确地说，不相等）发现了被赋的值：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Nullable Data *****\n");
  DatabaseReader dr = new DatabaseReader();

  // 从“数据库”获取int
  int? i = dr.GetIntFromDatabase();
  if (i.HasValue)
    Console.WriteLine("Value of 'i' is: {0}", i.Value);
  else
    Console.WriteLine("Value of 'i' is undefined.");

  // 从“数据库”获取bool
  bool? b = dr.GetBoolFromDatabase();
  if (b != null)
    Console.WriteLine("Value of 'b' is: {0}", b.Value);
  else
    Console.WriteLine("Value of 'b' is undefined.");
  Console.ReadLine();
}
```

4.6.2 ??操作符

关于可空类型需要知道的最后一点是，可以使用??操作符。在获得的值实际上是null时，我们可以用这个操作符给一个可空类型赋值。对于上面这个示例来说，假设从GetIntFromDatabase()返回的值是null（当然，这个方法被编程为总是返回null，但我相信你能明白其中的思路）：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Nullable Data *****\n");
  DatabaseReader dr = new DatabaseReader();
...
  // 从GetIntFromDatabase()返回的值为null时，将本地变量赋值为100
  int myData = dr.GetIntFromDatabase() ?? 100;
  Console.WriteLine("Value of myData: {0}", myData);
  Console.ReadLine();
}
```

使用??操作符的好处是，它比传统的if/else条件的写法更加紧凑。不过，如果愿意，你也可以编写如下功能相同的代码，以确保如果值为空，则设置为100：

```
// 使用 ? : ?? 语法的长版本
int? moreData = dr.GetIntFromDatabase();
if (!moreData.HasValue)
    moreData = 100;
Console.WriteLine("Value of moreData: {0}", moreData);
```

源代码 NullableTypes项目的源代码位于Chapter 4的子目录下。

4.7 小结

本章首先介绍了几个可用来构建自定义方法的C#关键字。还记得吗，在默认情况下，参数按值传递。然而，如果参数被标记为ref或out，我们可以按引用进行传递。我们还学习了可选参数的作用，以及如何定义和调用接受参数数组的方法。

在研究了方法重载的主题之后，接下来的大部分篇幅探讨了有关数组、枚举和结构如何在C#中定义，以及如何在.NET基础类库中进行表示。然后，我们研究了有关值类型和引用类型的细节，包括当作为参数传入方法后它们如何响应，以及如何使用?和??操作符来和可空数据类型进行交互。

Part 3 第三部分

C# 面向对象编程

本部分内容

第 5 章

封　　装

在前两章中，我们研究了所有.NET应用程序共有的许多核心语法结构。本章中，我们会研究C#的面向对象功能。首先介绍如何构建支持任意数量的构造函数的定义明确的类类型。理解了定义类以及分配对象的基本知识之后，本章余下的内容会研究封装的作用。然后我们会讨论如何定义类属性以及静态成员、对象初始化语法、只读字段、常量和分部类的作用。

5.1　C#类类型

就.NET平台而言，最基本的编程结构就是类类型。正式地说，类是由字段数据（通常叫做成员变量）以及操作这个数据的成员（如构造函数、属性、方法、事件等）所构成的自定义类型。总地来说，其中的字段数据用于表示类实例的“状态”（或称为对象）。C#这类面向对象的语言的强大之处就在于，通过将数据和相关功能集合在类定义中，我们就可以仿照现实生活中的实体来设计软件。

首先，新建一个名为SimpleClassExample的C#控制台应用程序。如图5-1所示，通过Project→Add Class菜单选项，从结果对话框中选择Class图标，单击Add按钮，插入一个新的类文件（名为Car.cs）到我们的项目中。

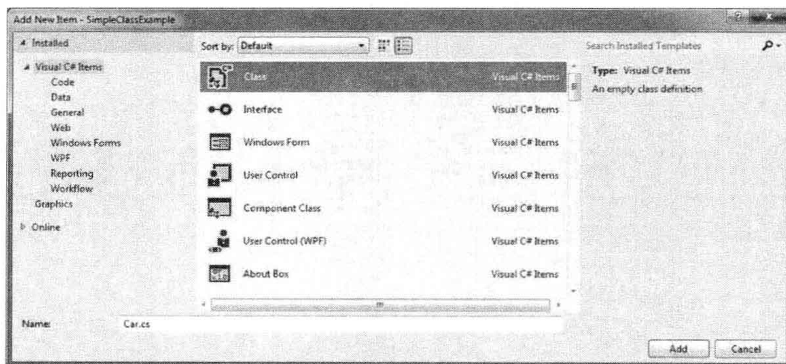


图 5-1　插入一个新的 C#类类型

在C#中，类使用class关键字来定义。下面是最简单的声明：

```
class Car
{
}
```

定义了类类型之后，我们需要考虑用于表示类状态的一组成员变量。例如，我们可能决定让Car维护一个int数据类型来表示当前速度，以及用一个string数据类型来表示汽车的昵称。有了这样的初始设计方案，按如下所示更新Car类：

```
class Car
{
  // Car的“状态”
  public string petName;
  public int currSpeed;
}
```

注意，这些成员变量都使用public访问修饰符来声明。类型的对象创建之后，就可以直接访问类的公共成员。你可能已经知道，“对象”这个术语表示使用new关键字创建的某个类类型的实例。

5

说明 类的字段数据很少定义为公共的。为了保护状态数据的完整性，最好将数据定义为私有的（或者是受保护的），并且通过类型属性（本章稍后会介绍）对数据提供受控制的访问。但为了让第一个示例足够简单，公共的数据正好符合这个要求。

定义了表示类状态的一组成员变量之后，下一个设计步骤就是创建描述其行为的成员。例如，Car类会定义一个名为SpeedUp()的方法和另一个名为PrintState()的方法，如下所示更新你的类：

```
class Car
{
  // Car的“状态”
  public string petName;
  public int currSpeed;

  // Car的功能
  public void PrintState()
  {
    Console.WriteLine("{0} is going {1} MPH.", petName, currSpeed);
  }

  public void SpeedUp(int delta)
  {
    currSpeed += delta;
  }
}
```

我们可以看到，PrintState()相当于一个调试功能，它只将某个Car对象的当前状态转储到命令窗口。SpeedUp()会根据传入的int参数指定的数量来增加Car的速度。现在，使用如下代码来更新Program类中的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");

  // 分配和设置Car对象
  Car myCar = new Car();
  myCar.petName = "Henry";
  myCar.currSpeed = 10;

  // 将Car加速几次，然后输出新的状态
```

```
    for (int i = 0; i <= 10; i++)
    {
      myCar.SpeedUp(5);
      myCar.PrintState();
    }
    Console.ReadLine();
  }
```

运行程序之后，我们会看到Car变量（myCar）在应用程序的整个生命中都维持当前的状态，如下述代码所示。

```
***** Fun with Class Types *****

Henry is going 15 MPH.
Henry is going 20 MPH.
Henry is going 25 MPH.
Henry is going 30 MPH.
Henry is going 35 MPH.
Henry is going 40 MPH.
Henry is going 45 MPH.
Henry is going 50 MPH.
Henry is going 55 MPH.
Henry is going 60 MPH.
Henry is going 65 MPH.
```

使用new关键字来分配对象

从之前的代码示例中我们可以看到，对象必须使用new关键字来分配到内存中。如果我们不使用new关键字，并且在之后的代码语句中尝试使用类变量的话，就会收到一个编译器错误。例如，下面的Main()方法不会编译：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");

  // 错误！忘记使用new创建对象
  Car myCar;
  myCar.petName = "Fred";
}
```

要使用new关键字正确创建对象，我们可以在单行代码中定义并分配一个Car对象：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
  Car myCar = new Car();
  myCar.petName = "Fred";
}
```

或者，如果我们希望分开定义和分配对象的话，可以这么做：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
  Car myCar;
```

```
    myCar = new Car();
    myCar.petName = "Fred";
  }
```

在这里，第一句代码只是声明了指向尚未被创建的Car对象的引用。在我们通过new关键字把引用赋给对象之后，这个引用才会指向内存中的有效类实例。

不管怎么样，在这里我们已经有了一个没什么实际意义的类，并且定义了一些数据和基本方法。要增强当前Car类的功能，我们需要理解*构造函数*的作用。

5.2 构造函数

由于对象有状态（由对象的成员变量的值来表示），对象用户通常希望在使用对象之前先给对象的字段数据赋相关的值。现在，Car类型需要petName和currSpeed字段逐一被赋值。对于当前示例来说，问题不大，因为我们只有两个公共的数据点。然而，一个拥有很多字段的类也是常见的。显然，要编写20个初始化语句来设置20个数据点是很麻烦的事情。

还好，C#支持*构造函数*，它允许在创建对象时创建其状态。构造函数是类的特殊方法，在使用new关键字创建对象时被间接调用。然而，和"普通"方法不同，构造函数永远不会返回值（即使是void），并且它的名字总是和需要构造的类的名字相同。

5.2.1 默认构造函数的作用

每一个C#类都提供了内建的*默认构造函数*，需要时可以重新定义。根据定义，默认的构造函数不会接受参数。除了把新对象分配到内存中，默认构造函数确保所有字段数据都设置为正确的默认值（更多有关C#数据类型默认值的信息，请参阅第3章）。

如果你对这些默认的赋值不满意，可以重新定义默认的构造函数来满足需求。按如下所示更新C# Car类来进行说明：

```
class Car
{
  // Car的状态
  public string petName;
  public int currSpeed;

  // 自定义的默认构造函数
  public Car()
  {
    petName = "Chuck";
    currSpeed = 10;
  }
...
}
```

在这里，我们强制所有Car对象一开始就命名为Chuck，时速为10。这样，我们就可以创建一个具有如下默认值的Car对象：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
```

```
    // 调用默认构造函数
    Car chuck = new Car();

    // 输出"Chuck is going 10 MPH."
    chuck.PrintState();
  ...
  }
```

5.2.2 定义自定义的构造函数

大多数情况下，除了默认构造函数之外，类还会定义其他构造函数。这样，我们就可以为对象用户提供简单而一致的方式：在创建对象时直接初始化对象的状态。考虑如下Car类的修改，现在它一共支持3个类构造函数：

```
class Car
{
  // Car的“状态”
  public string petName;
  public int currSpeed;

  // 自定义的默认构造函数
  public Car()
  {
    petName = "Chuck";
    currSpeed = 10;
  }

  // 在这里，currSpeed会获得int的默认值0
  public Car(string pn)
  {
    petName = pn;
  }

  // 让调用者设置Car的完整“状态”
  public Car(string pn, int cs)
  {
    petName = pn;
    currSpeed = cs;
  }
...
}
```

记住，让构造函数彼此不同（就C#编译器而言）的是构造函数参数的个数和类型。回顾一下第4章，当我们定义了具有同样名字但参数数量和类型不同的方法时，就是重载方法。因此，Car类型重载了构造函数来提供许多种在声明时创建对象的方式。不管怎么样，我们现在可以使用其中任何一种公共构造函数来创建Car对象。例如：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");

  // 创建一个叫Chuck的Car，时速为10MPH
  Car chuck = new Car();
  chuck.PrintState();

  // 创建一个叫Mary的Car，时速为0MPH
  Car mary = new Car("Mary");
```

```
  mary.PrintState();

  // 创建一个叫Daisy的Car，时速为75MPH
  Car daisy = new Car("Daisy", 75);
  daisy.PrintState();
...
}
```

5.2.3 再谈默认构造函数

我们已经知道了，所有类都有免费的默认构造函数。因此，如果我们插入一个新类Motorcycle到当前的项目中，并这么定义：

```
class Motorcycle
{
  public void PopAWheely()
  {
    Console.WriteLine("Yeeeeeee Haaaaaeewww!");
  }
}
```

就可以直接通过默认构造函数来创建Motorcycle类型的实例了：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
  Motorcycle mc = new Motorcycle();
  mc.PopAWheely();
...
}
```

然而，一旦定义了自定义构造函数，默认构造函数就被自动从类中移除，并且不再有效！这样来想吧，如果不定义自定义构造函数，C#编译器就会给我们一个默认值，以便对象用户分配类型实例，字段数据都设置为正确的默认值。然而，如果我们定义了唯一的构造函数，编译器就会认为我们会自己处理。

因此，如果希望对象用户使用默认构造函数和自定义构造函数创建类型实例，就必须显式重新定义默认构造函数。最后，请记住在大多数情况下，类的默认构造函数的实现故意为空，因为我们需要的只是创建具有默认值的对象的能力。考虑如下对Motorcycle类的更新：

```
class Motorcycle
{
  public int driverIntensity;

  public void PopAWheely()
  {
    for (int i = 0; i <= driverIntensity; i++)
    {
      Console.WriteLine("Yeeeeeee Haaaaaeewww!");
    }
  }
  // 恢复默认的构造函数，将所有数据成员设为默认值
  public Motorcycle() {}

  // 自定义构造函数
  public Motorcycle(int intensity)
  {
```

5

```
      driverIntensity = intensity;
    }
  }
```

说明　你应该已经能较好地理解类构造函数的作用了。下面我们来看个简便方法。Visual Studio IDE提供了ctor代码段，输入ctor并按下Tab键两次，IDE会自动定义一个定制的默认构造函数。此后你就可以添加自定义参数和实现逻辑了，你不妨尝试一下！

5.3　this 关键字的作用

C#支持this关键字来提供对当前类实例的访问。this关键字可能的用途就是，解决当传入参数的名字和类型数据字段的名字相同时产生的作用域歧义。当然，最理想的是我们采用一个不会产生这种歧义的命名习惯。然而，为了演示this关键字的使用，更新Motorcycle类，创建新的string字段（名为name）来表示司机的名字。然后，增加一个名为SetDriverName()的方法，其实现如下所示：

```
class Motorcycle
{
  public int driverIntensity;

  // 表示司机名称的新成员
  public string name;
  public void SetDriverName(string name)
  {
    name = name;
  }
...
}
```

尽管这段代码可以通过编译，但是Visual Studio会显示一条警告信息通知你使用变量本身来设置该变量。为了演示，更新Main()方法来调用SetDriverName()，然后输出name字段的值，你可能会惊奇地发现name字段的值是空字符串！

```
// 创建一个摩托车对象，其驾驶者为Tiny
Motorcycle c = new Motorcycle(5);
c.SetDriverName("Tiny");
c.PopAWheely();
Console.WriteLine("Rider name is {0}", c.name); // 输出空字符串
```

问题就是，由于编译器会认为name指向当前方法作用域内的变量，而不是类作用域中的name字段，因此SetDriverName()的实现为传入参数本身赋值了。要想让编译器知道我们希望将当前对象的name数据字段设置为传入的name参数，只需要使用this就能解决这个歧义：

```
public void SetDriverName(string name)
{
  this.name = name;
}
```

要知道，如果没有歧义的话，我们不需要在类访问它自己的数据或成员时使用this关键字。例如，如果我们重命名string数据成员为driverName（也需要更新Main()方法），this就是可选的，并且也不会存在作用域歧义：

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  public void SetDriverName(string name)
  {
    // 这两个语句从功能上说是一样的
    driverName = name;
    this.driverName = name;
  }
...
}
```

尽管在无歧义的情况下使用this，this也没有给我们很多惊喜，但实现成员时我们会发现这个关键字很有用，如SharpDevelop和Visual Studio这样的IDE在指定this时会启用智能感知。这样，如果我们忘记类成员的名字并且希望快速回忆起其定义，this就会很有用。考虑图5-2。

5



图5-2 this的智能感知

5.3.1 使用this进行串联构造函数调用

this关键字的另一种用法是使用一项名为构造函数链的技术来设计类。当类定义了多个构造函数时，这个设计模式就会很有用。由于构造函数通常会验证传入的参数来强制各种业务规则，所以在类的构造函数集合中经常会找到冗余的验证逻辑。考虑如下更新后的Motocycle：

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  public Motorcycle() { }

  // 冗余的构造函数逻辑
```

```
    public Motorcycle(int intensity)
    {
      if (intensity > 10)
      {
        intensity = 10;
      }
      driverIntensity = intensity;
    }

    public Motorcycle(int intensity, string name)
    {
      if (intensity > 10)
      {
        intensity = 10;
      }
      driverIntensity = intensity;
      driverName = name;
    }
  ...
  }
```

在这里（可能是要确保驾驶者的安全），每一个构造函数确保强度等级不超过10。虽然可以这么做，但是在两个构造函数中有冗余代码语句。这不够完美，如果规则改变的话，就必须在多个位置更新代码（例如，如果强度应该不大于5）。

改进这种情况的一个方法就是在Motorcycle类中定义一个用来验证传入参数的方法。如果这么做的话，每一个构造函数就可以在进行字段赋值之前调用这个方法。虽然这个方法确实可以隔离在业务规则改变时需要修改的代码，但是我们就会面临如下的冗余：

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  // 构造函数
  public Motorcycle() { }

  public Motorcycle(int intensity)
  {
    SetIntensity(intensity);
  }

  public Motorcycle(int intensity, string name)
  {
    SetIntensity(intensity);
    driverName = name;
  }

  public void SetIntensity(int intensity)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
  }
...
}
```

一个更简洁的方案就是，让一个接受最多参数个数的构造函数做“主构造函数”，并且实现必需的验证逻辑。其余的构造函数可以使用this关键字把传入的参数转发给主构造函数，并且提供所有必需的其他参数。这样，整个类中只会有一个构造函数需要我们去操心，其余构造函数基本上都是空的。

下面是Motorcycle类的最后一次迭代（为了演示，又多了一个构造函数）。在串联构造函数时，请注意this如何在构造函数本身的作用域之外“躲开”构造函数的声明（通过冒号操作符）：

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  // 构造函数链
  public Motorcycle() {}
  public Motorcycle(int intensity)
    : this(intensity, "") {}
  public Motorcycle(string name)
    : this(0, name) {}

  // 这是做所有工作的“主”构造函数
  public Motorcycle(int intensity, string name)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
...
}
```

需要理解的是，使用this关键字串联构造函数不是强制的。但如果使用这项技术，类定义就会更容易维护、更简明。再说一次，使用这项技术可以简化编程任务，因为真正的工作都交给了一个构造函数（通常这个构造函数有大多数的参数）来做，而其他构造函数只是将“皮球”踢给它就可以了。

说明 回想一下，第4章我们介绍过，C#支持可选参数，如果在类构造函数中使用可选参数，可以达到跟使用构造函数链一样的效果，还能省下不少代码。过会儿我们就来展示一下。

5.3.2 观察构造函数流程

最后，我们要知道在构造函数传递参数给指定的主构造函数（并且构造函数处理了数据）之后，调用者最初调用的构造函数还会执行所有剩余的代码语句。为了说明清楚，更新每一个Motorcycle类的构造函数，各自调用Console.WriteLine()：

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  // 构造函数链
  public Motorcycle()
```

```
  {
    Console.WriteLine("In default ctor");
  }

  public Motorcycle(int intensity)
    : this(intensity, "")
  {
    Console.WriteLine("In ctor taking an int");
  }

  public Motorcycle(string name)
    : this(0, name)
  {
    Console.WriteLine("In ctor taking a string");
  }
  // “主”构造函数完成所有实际工作
  public Motorcycle(int intensity, string name)
  {
    Console.WriteLine("In master ctor ");
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
...
}
```

现在，确保Main()方法按如下代码使用Motorcycle对象：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with class Types *****\n");

  // 创建Motorcycle
  Motorcycle c = new Motorcycle(5);
  c.SetDriverName("Tiny");
  c.PopAWheely();
  Console.WriteLine("Rider name is {0}", c.driverName);
  Console.ReadLine();
}
```

考虑前面的Main()方法输出：

```
***** Fun with class Types *****

In master ctor
In ctor taking an int
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Yeeeeeee Haaaaaeewww!
Rider name is Tiny
```

我们可以看到，构造函数的逻辑流程如下。

- 通过调用只有单个int的构造函数来创建对象。
- 构造函数将提供的数据转发给主构造函数，并且提供调用者没有提供的其他初始参数。
- 主构造函数把传入的数据赋值给对象的字段数据。
- 控制返回到最初调用的构造函数，并且执行所有剩余的代码语句。

使用构造函数链的好处是，这种编程模式在任何版本的C#语言和.NET平台中都是可用的。然而，在使用.NET 4.0或更高版本时，你还可以使用可选参数代替传统的构造函数链，从而大大简化编程任务。

5.3.3 再谈可选参数

我们在第4章中学习了可选参数和命名参数。可选参数允许我们对传入参数提供默认值。如果调用者希望使用这些默认值而不是使用自定义数据，就不必再单独指定这些参数。考虑下面这个版本的Motorcycle类，它只用一个构造函数定义却提供了多种构造对象的方式：

```
class Motorcycle
{
  // 使用可选参数的单个构造函数
  public Motorcycle(int intensity = 0, string name = "")
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
...
}
```

有了这个构造函数，现在你可以分别使用0个、1个或2个参数来新建Motorcycle对象。使用命名参数语法，可以跳过接受的默认设置（见第3章）。

```
static void MakeSomeBikes()
{
  // driverName = "", driverIntensity = 0
  Motorcycle m1 = new Motorcycle();
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m1.driverName, m1.driverIntensity);

  // driverName = "Tiny", driverIntensity = 0
  Motorcycle m2 = new Motorcycle(name:"Tiny");
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m2.driverName, m2.driverIntensity);

  // driverName = "", driverIntensity = 7
  Motorcycle m3 = new Motorcycle(7);
  Console.WriteLine("Name= {0}, Intensity= {1}",
    m3.driverName, m3.driverIntensity);
}
```

尽管可选/命名参数灵巧地简化了为给定类定义构造函数集的方式，但要记住的是这种语法只能在.NET 4.0或更高版本的平台下运行。如果你需要构建一个在任何版本的.NET平台下都能工作的类，最好坚持使用传统的构造函数链技术。

不管怎样，现在我们已经可以定义一个包含字段数据（即成员变量）和各种成员（如方法和构造函数）的类了。接下来，我们正式介绍static关键字。

源代码　SimpleClassExample项目的源代码位于Chapter 5子目录下。

5.4 static 关键字

C#类（或结构）可以通过static关键字来定义许多静态成员。如果这样的话，这些成员就只能直接从类级别而不是对象引用调用。为了说明区别，考虑一下我们的老朋友System.Console。可以看到，我们没有从对象级别调用WriteLine()方法：

```
// 错误！WriteLine()不是对象级别的方法
Console c = new Console();
c.WriteLine("I can't be printed...");
```

而是将类型名字作为静态成员WriteLine()的前缀：

```
// 正确！WriteLine()是静态方法
Console.WriteLine("Thanks...");
```

简而言之，静态方法被（类设计者）认为是非常普遍的项，并且不需要在调用成员时创建类型的实例。虽然任何类都可以定义静态方法，但是它们通常出现在“工具类”中。根据定义，工具类是不维护任何对象级别的状态且并非由new关键字创建的类。因此，工具类会以类级别（即静态）成员公开所有功能。

例如，如果我们使用Visual Studio对象浏览器（通过View→Object Browser菜单项）来查看mscorlib.dll的System命名空间，你将发现Console、Math、Environment和GC类的所有成员都通过静态成员公开其所有功能。.NET基础类库中的工具类不多，有几个。

另外，还要注意，静态成员并不只是存在于工具类中，它们可以是任何类定义的一部分。只需记住静态成员推动给定项到类级别，而不是对象级别。在后面的几节中我们会看到，static关键字可应用于：

- 类的数据
- 类的方法
- 类的属性
- 构造函数
- 整个类定义

让我们从静态数据的概念开始，看看每种情况。

说明　本章后面在学习属性的作用时你就能一睹静态属性的作用了。

5.4.1 定义静态数据

当我们设计类时，大部分情况下，我们会将数据定义为实例级别的数据，也就是非静态数据。如果类定义了非静态数据（或者说是实例数据），类型的每一个对象都会维护字段的独立副本。相较而

言，定义类的静态数据时，同一类别的所有对象都会共享内存。

```
//一个简单的储蓄账户类
class SavingsAccount
{
  // 实例级别的数据
  public double currBalance;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
}
```

创建SavingsAccount对象的时候，每一个对象都会分配currBalance字段的内存。因此你可以创建五个不同的SavingsAccount对象，每一个都有自己的余额。这样，改变一个账户的余额，其他对象都不会受影响。

在另一方面，静态数据分配一次并且在相同类型的所有对象之间共享。为了演示静态数据的作用，为SavingsAccount类增加一个名为currInterestRate的静态数据点，并设置为默认值0.04：

```
//一个简单的储蓄账户类
class SavingsAccount
{
  //实例级别的数据
  public double currBalance;

  //静态数据点
  public static double currInterestRate = 0.04;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
}
```

如果要在Main()方法中创建3个SavingsAccount实例，可以按如下方式编写代码：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Data *****\n");
  SavingsAccount s1 = new SavingsAccount(50);
  SavingsAccount s2 = new SavingsAccount(100);
  SavingsAccount s3 = new SavingsAccount(10000.75);
  Console.ReadLine();
}
```

内存中的数据分配和图5-3所示的差不多。

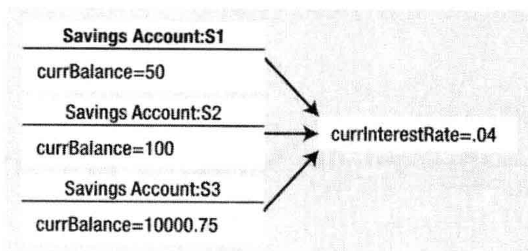


图5-3 静态数据分配一次并被类的所有实例所共享

这里，我们的前提是所有储蓄账户的利率都相同。因为静态数据字段是由所有对象共享的，因此如果以某种方式改变了它，所有对象在下次访问静态数据时都会看到新值。因为它们实际上都关注同一个内存位置。要了解如何改变（或获取）静态数据，需要考虑静态方法的作用。

5.4.2 定义静态方法

下面更新SavingsAccount类来定义两个静态方法。第一个静态方法（GetInterestRate()）会返回当前利率，第二个静态方法（SetInterestRate()）允许你改变利率。

```
// 一个简单的储蓄账户类
class SavingsAccount
{
  // 实例级别数据
  public double currBalance;

  // 静态数据点
  public static double currInterestRate = 0.04;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }

  // 获取/设置利率的静态成员
  public static void SetInterestRate(double newRate)
  { currInterestRate = newRate; }

  public static double GetInterestRate()
  { return currInterestRate; }
}
```

现在，看看下面的用法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Data *****\n");
  SavingsAccount s1 = new SavingsAccount(50);
  SavingsAccount s2 = new SavingsAccount(100);

  // 打印当前利率
  Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

  // 生成新对象，这并没有重设利率
  SavingsAccount s3 = new SavingsAccount(10000.75);
  Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

  Console.ReadLine();
}
```

前面的Main()方法的输出如下所示：

```
***** Fun with Static Data *****

Interest Rate is: 0.04
Interest Rate is: 0.04
```

可以看到，当创建SavingsAccount类的新实例时，静态数据的值没有重置，因为CLR把数据分配到内存中只会进行一次。之后，所有SavingsAccount类型的对象会针对currInterestRate字段操作同一个值。

在设计C#类时，哪一部分数据应该定义为静态成员，而哪一部分数据不应该定义为静态成员，这是一个难题。尽管没有硬性规定，但要记住静态数据字段是由所有对象共享的。因此，如果你要定义一个所有对象都可以分享的数据点，就可以使用静态成员。

想想看如果利率变量不定义成静态的，将会是什么样。这意味着每个SavingsAccount对象都必须包含自己的currInterestRate字段。假设你创建了100个SavingsAccount对象，并且需要改变利率。那么就需要调用100次SetInterestRate()方法！这显然不是建立共享数据模型的有效方法。同样，如果同种类的所有对象都经常使用某个值，也非常适合使用静态数据。

说明 静态成员在其实现中引用非静态成员会导致编译器错误。与此类似，在静态成员中将this操作符用做暗示对象的“this”也是错误的。

5.4.3 定义静态构造函数

在本章前面解释过，常见的构造函数用于在创建对象时设置对象的实例数据的值。因此，如果在实例级别的构造函数中赋值给静态数据成员，你会惊奇地发现每次新建对象的时候，值都会重置。例如，假设我们按如下所示更新SavingsAccount类（同时注意我们不再内联初始化currInterestRate字段）：

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate;

  //注意构造函数正在设置静态成员currInterestRate的值
  public SavingsAccount(double balance)
  {
    currInterestRate = 0.04;  //这是一个静态数据
    currBalance = balance;
  }
...
}
```

在Main()方法中添加以下代码

```
static void Main( string[] args )
{
  Console.WriteLine("***** Fun with Static Data *****\n");

  //创建一个账户
  SavingsAccount s1 = new SavingsAccount(50);

  //输出当前利率
  Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());

  //通过属性改变利率
  SavingsAccount.SetInterestRate(0.08);

  //再创建一个账户
  SavingsAccount s2 = new SavingsAccount(100);
```

```
    //应该打印0.08，对吗
    Console.WriteLine("Interest Rate is: {0}", SavingsAccount.GetInterestRate());
    Console.ReadLine();
  }
```

如果我们执行之前的Main()方法，就会注意到，每次创建新的SavingsAccount对象的时候，currInterestRate变量就会被重置，并且总是被设置为0.04。显然，用一个普通的实例级别的构造函数设置静态数据的值，这在某种程度上违背了我们最初的意愿。每次新建对象，类级别的数据都将被重置！设置静态字段的一种方法是像原来一样使用成员初始化语法：

```
class SavingsAccount
{
  public double currBalance;

  // 静态数据点
  public static double currInterestRate = 0.04;
...
}
```

不管创建多少个对象，这种方法都可以确保静态数据只被分配一次。但是，万一静态数据的值是在运行时获取的呢？例如，在一个普通的银行应用程序中，利率变量的值应该从数据库或外部文件中读取。要完成这样的任务，需要一个类似构造函数的方法作用域来执行代码语句。

正是由于这个原因，C#允许我们定义静态构造函数，安全地设置静态数据的值。考虑下面的更新：

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }

  // 静态构造函数
  static SavingsAccount()
  {
    Console.WriteLine("In static ctor!");
    currInterestRate = 0.04;
  }
...
}
```

简而言之，静态构造函数是特殊的构造函数，并且非常适用于初始化在编译时未知的静态数据的值（例如，我们需要从外部文件读取值或者生成随机数等）。假如你重新运行之前的Main()方法，输出会如你所料。注意信息“In static ctor!”只会打印一次，因为CLR在第一次使用之前调用了所有静态构造函数（且对于应用程序的该实例不再调用它们）：

```
***** Fun with Static Data *****

In static ctor!
Interest Rate is: 0.04
Interest Rate is: 0.08
```

这里是有关静态构造函数有趣的几点。

- 一个类只可以定义一个静态构造函数。换句话说，就是静态构造函数不能被重载。
- 静态构造函数不允许访问修饰符并且不能接受任何参数。
- 无论创建了多少类型的对象，静态构造函数只执行一次。
- 运行库创建类实例或调用者首次访问静态成员之前，运行库会调用静态构造函数。
- 静态构造函数的执行先于任何实例级别的构造函数。

有了这样的修改，在我们创建新SavingsAccount对象时，静态数据的值就会被维持。因为不管创建了多少对象，静态成员只会在静态构造函数中设置一次。

源代码 StaticDataAndMember项目包含在Chapter 5子目录下。

5.4.4 定义静态类

也可以直接在类级别应用static关键字。如果一个类被定义为静态的，就不能使用new关键字来创建，并且只能包含用static关键字标记的成员或字段。如果不是这样，就会收到编译错误。

说明 只包含静态功能的类或结构通常称为工具类。在设计工具类时，将类定义为静态类是一个非常好的做法。

乍一看这好像是完全无用的特性，因为一个类不能被创建似乎不算什么好处。然而，如果我们创建一个只包含静态成员或常量数据的类，就不需要先进行分配。为了说明这一点，我们创建一个控制台应用程序SimpleUtilityClass，下一步定义一个新的静态类类型：

```
// 静态类只能包含静态成员
static class TimeUtilClass
{
  public static void PrintTime()
  { Console.WriteLine(DateTime.Now.ToShortTimeString()); }

  public static void PrintDate()
  { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}
```

由于这个类定义了static关键字，我们就不能使用new关键字创建TimeUtilClass的实例，而是所有功能都从类级别公开：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Data *****\n");

  // 这刚刚好
  TimeUtilClass.PrintDate();
  TimeUtilClass.PrintTime();
  // 编译器错误！不能创建静态类
  TimeUtilClass u = new TimeUtilClass ();

  Conskle.ReadLine();
}
```

本章进行至此，你应该已经对创建包含构造函数、字段以及各种静态（非静态）成员的类类型有了充分的了解。既然已经熟悉了这些基础，那么让我们来正式研究OOP（面向对象编程）的3个支柱。

源代码 SimpleUtilityClass项目的源代码位于Chapter 5子目录下。

5.5 定义 OOP 的支柱

所有面向对象的语言（C#、Java、C++和Visual Basic等）必须满足OOP的3个核心原则，通常也叫做“OOP的支柱”。

- **封装**：这种语言怎样隐藏一个对象的内部实现并且保护数据完整性？
- **继承**：这种语言是怎样促进代码重用的？
- **多态**：这种语言是怎样让你用同样的方式处理相关对象的？

在深入每一个支柱的语法细节之前，理解每一个支柱的基本作用是很重要的。因此，下面简要介绍一下，本章其余部分和第6章将会讨论完整细节。

5.5.1 封装的作用

OOP的第一个支柱是封装（encapsulation）。这是将对象用户[①]不必了解的实现细节隐藏起来的一种语言能力。例如，假设你正在使用DatabaseReader类，它有Open()和Close()两个方法：

```
// 假设DatabaseReader封装了数据库操作的细节
DatabaseReader dbReader = new DatabaseReader();
dbReader.Open(@"C:\AutoLot.mdf");

// 使用数据文件来做一些事情，然后关闭文件
dbReader.Close();
```

这个假想的DatabaseReader类封装了查找、加载、操作和关闭数据文件的内部细节。程序员喜欢封装，因为OOP的这个支柱让程序设计任务更简单。没有必要担心在幕后完成DatabaseReader类的工作的众多代码，只需创建一个实例并发送合适的消息（例如，“打开位于C驱动器的AutoLot.mdf文件”）即可。

和封装编程逻辑紧密相关的概念是数据保护的概念。理想情况下，对象的状态数据应该使用private（或protected）关键字来指定。这样的话，外部世界就不能直接改变或获取底层的值，而必须“有礼貌地请求”。这是好事情，因为公共的数据点很容易被破坏（但愿是无意的，而不是有意的）。我们稍后会全面研究封装。

5.5.2 继承的作用

OOP的下一个支柱是继承（inheritance），它是指基于已有类定义来创建新类定义的语言能力。本质上，通过继承，子类可以继承基类（或称父类）核心的功能，并扩展基类的行为。图5-4是一个简单的例子。

---

① 指使用你编写的类进行编程的人。——编者注

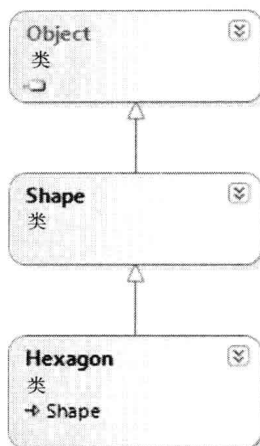


图5-4 is-a关系

从图5-4中我们可以知道"六边形是图形，图形又是对象"。如果我们的类是这种形式的继承，就在类型之间创建一种"is-a"的关系。"is-a"关系又称为继承。

在这里，我们会认为Shape定义了许多所有派生类都公有的成员（可能是用于绘制形状的颜色值，也可能是表示高度和宽度的值）。由于Hexagon类扩展了Shape，它也就继承了由Shape和Object定义的核心功能，并且它自己也定义了其他六边形相关的功能（不管是什么）。

说明 在.NET平台下，System.Object是所有类层次关系中的最高父类，它定义了最基本的一些功能，在第6章中会完整介绍。

在OOP中还有另一种形式的代码重用：包含/委托模型（也就是has-a关系，或称聚合）。这种重用的形式不是用来建立父类/子类关系的。它意味着，一个类可以定义另一个类的成员变量，并向对象用户间接公开它的功能（如果需要的话）。

例如，给一辆汽车建模，你可能想表达一辆车has-a（有一个）收音机的概念。让Car继承Radio类或反之都是不合逻辑的。（Car是一个Radio？当然不是！）实际上，你有两个类一起合作，其中Car类创建并公开了Radio的功能：

```
class Radio
{
  public void Power(bool turnOn)
  {
    Console.WriteLine("Radio on: {0}", turnOn);
  }
}

class Car
{
  // 汽车"has-a"收音机
  private Radio myRadio = new Radio();

  public void TurnOnRadio(bool onOff)
```

```
    {
      // 到内部对象的委托调用
      myRadio.Power(onOff);
    }
  }
```

注意，对象用户不知道Car类在使用内部的Radio对象：

```
static void Main(string[] args)
{
    // 调用在内部被转发到Radio
  Car viper = new Car();
  viper.TurnOnRadio(false);
}
```

5.5.3　多态的作用

OOP的最后一个支柱是多态（polymorphism），它表示的是语言以同一种方式处理相关对象的能力。准确地说，这个面向对象语言的原则允许基类为所有的派生类定义一个成员集合（正式的术语为多态接口）。一个类类型的多态接口由任意个虚拟（virtual）或抽象（abstract）成员组成（第6章会详细介绍）。

简而言之，虚拟成员是定义默认实现的基类中的成员，它可能被派生类改变（更正式的说法是重写），而抽象方法是基类中不提供默认实现的成员，但它提供签名。当一个类派生自定义了抽象方法的基类时，抽象方法必须被派生类所重写。当派生类重写由基类定义的成员时，其实就重定义了响应相同请求的方式。

为了预览多态，让我们再来看图5-5中的图形层次结构。假设Shape类定义了Draw()虚拟方法，该方法不带参数。因为每一个图形需要用不同的方式绘制，所以子类（例如，Hexagon和Circle）可以按自己的需要重写这个方法（如图5-5所示）。

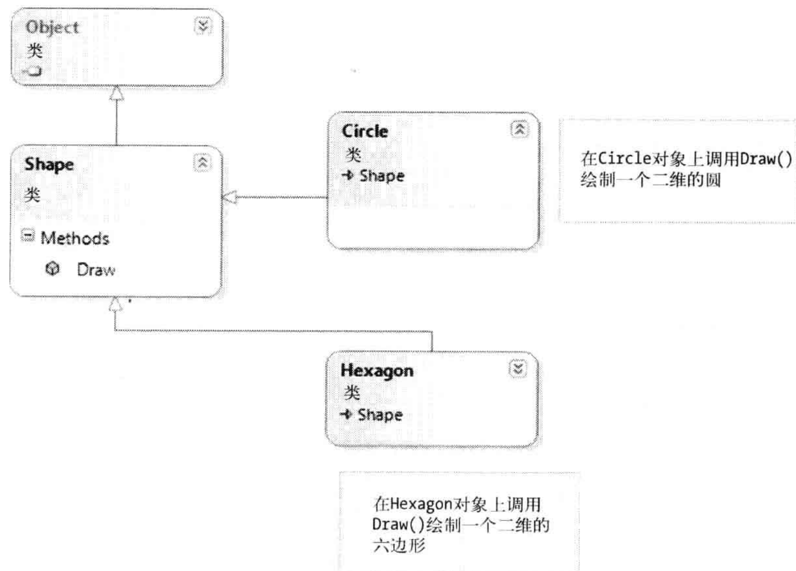


图5-5　传统多态

在设计完多态接口后，可以在代码中开始做不同的假设。例如，由于Hexagon和Circle都继承自共同的父类(Shape)，因此Shape类型的数组可以包含任何派生自该基类的类型。此外，由于Shape为所有派生类定义了多态接口（本例中为Draw()方法），我们可以假设数组中的每个成员都具有该功能。

考虑下面的Main()方法，它指示一个Shape的派生类型的数组使用Draw()方法绘制自己：

```
class Program
{
  static void Main(string[] args)
  {
    Shape[] myShapes = new Shape[3];
    myShapes[0] = new Hexagon();
    myShapes[1] = new Circle();
    myShapes[2] = new Hexagon();

    foreach (Shape s in myShapes)
    {
      // 使用多态接口
      s.Draw();
    }
    Console.ReadLine();
  }
}
```

我们对OOP支柱的基本讲述就是这些。现在你已经对这些理论有了概念，本章的其他篇幅会研究C#如何处理封装的更多细节。第6章会处理继承和多态的细节。

5.6 C#访问修饰符

在使用封装的时候，我们必须考虑类型的哪些方面对我们应用程序的哪些部分可见。准确地说，类型（类、接口、结构、枚举以及委托）以及它们的成员（属性、方法、构造函数、字段等）总是使用某个关键字来定义，这个关键字用来控制它们对应用程序其他部分如何“可见”。尽管C#定义了许多关键字来控制权限，但是它们会因为应用的地方不同（类型或成员）而不同。表5-1列举了每一个访问修饰符的作用以及它们可以应用到的地方。

表5-1 C#访问修饰符

C#访问修饰符	可以应用到的地方	作　用
Public	类型或者类型成员	公共的项没有限制。公共成员可从对象以及任何派生类访问。公共类型可以从其他外部程序集进行访问
private	类型成员或者嵌套类型	私有项只能由定义它们的类（或结构）进行访问
protected	类型成员或者嵌套类型	受保护项可以由定义它们的类及其任意子类使用，但外部类无法通过C#的点操作符访问
internal	类型或者类型成员	内部项只能在当前程序集中访问。因此，如果我们在.NET类库中定义一组内部类型的话，其他程序集就不能使用它们
protected internal	类型成员或者嵌套类型	如果在一个项上组合protected和internal关键字，项在定义它们的程序集、类以及派生类中可用

在本章中，我们只关心public和private关键字。后面的章节会研究internal和protected internal修饰符（构建.NET代码类库的时候有用）以及protected修饰符（在创建类层次关系的时候有用）的作用。

5.6.1　默认的访问修饰符

默认情况下，类型成员是隐式私有的，而类型是隐式内部的。因此，下面的类定义自动设置为内部的，而类型的默认构造函数自动设置为私有的：

```
// 具有私有默认构造函数的内部类
class Radio
{
  Radio(){}
}
```

因此，要允许其他类型调用对象的成员，我们必须将它们标记为公共可访问的。同样，如果我们希望对外部程序集（再说一次，在构建.NET代码库的时候很有用，见第14章）公开Radio，就需要增加public修饰符。

```
// 具有公共默认构造函数的公共类
public class Radio
{
  public Radio(){}
}
```

5.6.2　访问修饰符和嵌套类型

在表5-1中提到过，private、protected以及protected internal访问修饰符可以应用到嵌套类型上。第6章会详细研究嵌套。然而，在这里我们需要知道的是，嵌套类型是直接声明在类或结构作用域中的类型。下面的示例是一个嵌套在一个公共类（叫做SportsCar）中的私有枚举（名为CarColor）：

```
public class SportsCar
{
  // 没错！嵌套类型可以标记为私有的
  private enum CarColor
  {
    Red, Green, Blue
  }
}
```

在这里，我们可以在嵌套类型上应用private访问修饰符。然而，非嵌套类型（如SportsCar）只能用public或internal修饰符定义。因此，下面的类定义是不合法的：

```
// 错误！非嵌套类型不能被标记为私有的
private class SportsCar
{}
```

5.7 第一个支柱：C#的封装服务

封装概念的核心是，对象的内部数据不应该从对象实例直接访问。对象数据应该被定义为私有的，如果调用者想改变对象的状态，就要间接使用公共成员。为了解释封装的必要性，假设你创建了下面的类定义：

```
// 只有一个公共字段的类
class Book
{
  public int numberOfPages;
}
```

公共数据的问题是，它们无法知道，被赋予的当前值是否符合系统的当前业务规则。C#的int上限是非常大的（2 147 483 647）。因此，编译器允许下面的赋值：

```
// 嗯，是这样
static void Main(string[] args)
{
  Book miniNovel = new Book();
  miniNovel.numberOfPages = 30000000;
}
```

尽管没有溢出Int数据类型的边界，但很显然，一部有3000万页的迷你小说在现实世界中是不合理的。可见，公共字段不能提供设置上下限的方法。如果系统有一条业务规则规定一部迷你小说必须有1~1000页，怎样编程施加这样的规则就成了大问题。正因为这样，公共字段一般不会出现在产品级的类定义中。

说明 更具体地说，表示对象状态的类成员都不应该标记为公共的。在本章后面你将看到，可以使用公共常量和公共只读字段。

封装提供了一种保护状态数据完整性的方法。与定义公共字段相比（很容易发生数据损坏），应该更多地定义私有数据字段，这种字段可以由调用者间接地操作。定义私有字段的主要方式有以下两种：

- 定义一对传统的访问方法（get，也称为获取方法）和修改方法（set，也称为设置方法）；
- 定义一个.NET属性。

无论选择哪一种技术，关键是封装良好的类应该对外部世界隐藏操作数据方式的细节。这通常称为黑盒编程（black box programming）。这种方法的优势在于，对象可以自由地改变一个给定方法在底层的实现方式，而不影响任何已存在的使用它的代码（假设方法的参数和返回值保持不变）。

5.7.1 使用传统的访问方法和修改方法执行封装

在本章剩余的内容中，我们会构建一个相对完整的类来模拟普通员工。首先，新建一个名为EmployeeApp的控制台应用程序，然后使用Project→Add类菜单项插入一个新的类文件（叫做Employee.cs）。使用如下字段、方法以及构造函数更新Employee类：

```
class Employee
{
  // 字段数据
```

```
    private string empName;
    private int empID;
    private float currPay;

    // 构造函数
    public Employee() {}
    public Employee(string name, int id, float pay)
    {
      empName = name;
      empID = id;
      currPay = pay;
    }

    // 方法
    public void GiveBonus(float amount)
    {
      currPay += amount;
    }

    public void DisplayStats()
    {
      Console.WriteLine("Name: {0}", empName);
      Console.WriteLine("ID: {0}", empID);
      Console.WriteLine("Pay: {0}", currPay);
    }
  }
```

注意，现在使用private关键字来定义Employee类的字段。这样，就不能从对象变量直接访问empName、empID以及currPay字段。因此，Main()方法中的以下逻辑会导致编译器错误：

```
static void Main(string[] args)
{
  // 错误！不能直接从对象访问私有成员
  Employee emp = new Employee();
  emp.empName = "Marv";
}
```

如果希望外部世界和表示工人姓名的私有字符串进行交互，传统的做法是定义访问方法（即get方法）和修改方法（即set方法）。set方法可以改变当前实际状态数据的值，只要它满足定义的业务规则。

为了演示这一点，我们封装一个empName字段。向Employee类添加如下所示的public方法。注意，SetName()方法对传入的数据进行了检查，以确保string的字符数少于或等于15。否则，将在控制台打印错误并返回没有修改的empName字段。

说明　如果这是一个产品级的类，你还需要在构造函数逻辑中检查员工名称的字符长度。我们暂时忽略这个细节，等到学习.NET属性语法时再整理这些代码。

```
class Employee
{
  // 字段数据
  private string empName;
  ...

  // 访问方法（get方法）
  public string GetName()
  {
```

```
      return empName;
    }

    // 修改方法（set方法）
    public void SetName(string name)
    {
      // 在赋值之前检查输入的值
      if (name.Length > 15)
        Console.WriteLine("Error!  Name must be less than 16 characters!");
      else
        empName = name;
    }
  }
```

这个技术需要两个操作单个数据点的特殊方法。为了测试新方法，按如下所示更新Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
  Employee emp = new Employee("Marvin", 456, 30000);
  emp.GiveBonus(1000);
  emp.DisplayStats();

  // 使用get/set方法来和对象的名字进行交互
  emp.SetName("Marv");
  Console.WriteLine("Employee is named: {0}", emp.GetName());
  Console.ReadLine();
}
```

鉴于SetName()方法中的代码，如果你指定一个大于16个字符的字符串（如下所示），控制台将打印一个硬编码的错误消息：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
...
  //大于16个字符！控制台将打印错误消息
  Employee emp2 = new Employee();
  emp2.SetName("Xena the warrior princess");

  Console.ReadLine();
}
```

到目前为止，一切正常。我们用两个方法GetName()和SetName()封装了私有变量empName。如果要进一步封装Employee类中的数据，需要添加各种传统方法（如GetID()、SetID()、GetCurrentPay()、SetCurrentPay()）。每个修改方法都可能包含多行代码来执行额外的业务规则检查。虽然这没什么复杂的，但C#语言提供了一个可选的记法来封装类数据。

5.7.2 使用.NET属性进行封装

尽管可以使用传统的获取方法和设置方法封装这些字段数据，.NET语言还是提倡使用属性来强制数据封装状态数据。首先，理解属性总是映射到“实际的”访问方法和修改方法。因此，类的设计者还是可以在值赋值之前执行任何必要的内部逻辑（比如，对值进行大写转换，过滤值中的不合法字符，检查数字值的边界等）。

下面是更新后的Employee类，现在使用属性语法而不是传统的获取方法和设置方法来强制封装：

```
class Employee
{
  // 字段数据
  private string empName;
  private int empID;
  private float currPay;

  // 属性
  public string Name
  {
    get { return empName; }
    set
    {
      if (value.Length > 15)
        Console.WriteLine("Error!  Name must be less than 16 characters!");
      else
        empName = value;
    }
  }

  // 我们可以在这些属性的set中添加额外的业务规则，但本例中没有这个必要
  public int ID
  {
    get { return empID; }
    set { empID = value; }
  }
  public float Pay
  {
    get { return currPay; }
    set { currPay = value; }
  }
...
}
```

C#属性由属性作用域中定义的get作用域（访问方法）和set作用域（修改方法）构成。注意，属性通过返回值指定了它所封装的数据类型。还要注意的是，属性在定义时没有使用括号（甚至空括号）。考虑下面ID属性的注释：

```
// int表示该属性所封装的数据类型
// 数据类型必须与相关的字段（empID）相同
public int ID  // 没有括号
{
  get { return empID; }
  set { empID = value; }
}
```

在属性的set作用域中，我们使用了value标记，它用来表示调用者设置属性时传入的值。该标记不是真正的C#关键字，而是上下文关键字。在属性的set作用域里，value总是表示调用者设置的值，并且总是和属性本身的数据类型相同。因此，注意Name属性是如何测试string的范围的：

```
public string Name
{
  get { return empName; }
  set
  {
    // 这时，value是一个string
    if (value.Length > 15)
```

```
        Console.WriteLine("Error!  Name must be less than 16 characters!");
      else
        empName = value;
    }
  }
```

一旦创建了属性，对调用者来说就好像是获取或设置公共数据点，但在背后会调用相应的get或set块来保持封装：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
  Employee emp = new Employee("Marvin", 456, 30000);
  emp.GiveBonus(1000);
  emp.DisplayStats();

  // 设置和获取Name属性
  emp.Name = "Marv";
  Console.WriteLine("Employee is named: {0}", emp.Name);
  Console.ReadLine();
}
```

属性（和访问方法、修改方法相对）还能让我们的类型易于操作，因为属性可以结合C#内部操作符进行使用。假设我们的Employee类类型有一个私有成员变量表示员工的年龄。这里是相关更新（注意使用了构造函数链）：

```
class Employee
{
...
  // 新字段和属性
  private int empAge;
  public int Age
  {
    get { return empAge; }
    set { empAge = value; }
  }

  // 更新的构造函数
  public Employee() {}
  public Employee(string name, int id, float pay)
    :this(name, 0, id, pay){}

  public Employee(string name, int age, int id, float pay)
  {
    empName = name;
    empID = id;
    empAge = age;
    currPay = pay;
  }

  // 更新的DisplayStats()方法现在考虑了年龄
  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", empName);
    Console.WriteLine("ID: {0}", empID);
    Console.WriteLine("Age: {0}", empAge);
    Console.WriteLine("Pay: {0}", currPay);
  }
}
```

5

现在假设我们创建了名为joe的Employee对象。在他生日的时候，我们希望为其年龄加1。使用传统的访问方法和修改方法，需要如下代码：

```
Employee joe = new Employee();
joe.SetAge(joe.GetAge() + 1);
```

然而，如果使用Age属性封装empAge，就可以将代码简化为：

```
Employee joe = new Employee();
joe.Age++;
```

5.7.3　使用类的属性

属性，特别是属性的set部分，常用于打包类的业务规则。当前，Employee类的Name属性可以确保名称不多于15个字符。其余的属性（ID、Pay和Age）也可以用相关的逻辑进行更新。

尽管这很好，但是想想通常情况下类的构造函数会做些什么呢？接收传入参数，检查有效数据，然后赋值给内部私有字段。现在，主构造函数没有检查传入字符串数据的有效范围，因此可以这样修改：

```
public Employee(string name, int age, int id, float pay)
{
  // 嗯，这看上去可能有问题
  if (name.Length > 15)
    Console.WriteLine("Error!  Name must be less than 16 characters!");
  else
    empName = name;

  empID = id;
  empAge = age;
  currPay = pay;
}
```

我相信你能看出这种方法的问题所在。Name属性和主构造函数执行了相同的错误检查！如果对其他数据点也执行这样的检查，将产生大量的重复代码。为了简化代码，并把所有的错误检查隔离到一个中心位置，可以一直在类中使用属性，不管你何时需要获取或设置属性的值。考虑如下修改后的构造函数：

```
public Employee(string name, int age, int id, float pay)
{
  // 更好一些！在设置类数据的时候使用属性，减少了重复的错误检查
  Name = name;
  Age = age;
  ID = id;
  Pay = pay;
}
```

除了更新构造函数使用属性分配值外，在整个类实现中一直使用属性也是不错的做法，这可以确保业务规则总是被强制检查。在多数情况下，唯一需要直接使用私有数据的情况是在属性内部。考虑到这一点，用如下的代码更新Employee类：

```
class Employee
{
  // 字段数据
  private string empName;
```

```
    private int empID;
    private float currPay;
    private int empAge;

    // 构造函数
    public Employee() { }
    public Employee(string name, int id, float pay)
      :this(name, 0, id, pay){}
    public Employee(string name, int age, int id, float pay)
    {
      Name = name;
      Age = age;
      ID = id;
      Pay = pay;
    }

    // 方法
    public void GiveBonus(float amount)
    { Pay += amount; }

    public void DisplayStats()
    {
      Console.WriteLine("Name: {0}", Name);
      Console.WriteLine("ID: {0}", ID);
      Console.WriteLine("Age: {0}", Age);
      Console.WriteLine("Pay: {0}", Pay);
     }

    // 和前面例子一样的属性
  ...
  }
```

5

5.7.4 只读和只写属性

当封装数据时，可能希望配置一个只读属性。为此，可以忽略set块。类似地，如果想拥有一个只写属性，就忽略get语句块。在这个例子中我们没有必要这样做，下面的代码说明了SocialSecurityNumber属性（封装一个私有的字符串变量empSSN）是怎样被转换为只读的：

```
public string SocialSecurityNumber
{
  get { return empSSN; }
}
```

有了这个调整，设置雇员社会安全号码（SSN）的唯一方式就是通过构造函数参数。而如果试图在主构造函数中设置雇员的社会安全号码，将会收到编译器错误，因为该属性只读：

```
public Employee(string name, int age, int id, float pay, string ssn)
{
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
    // 啊，如果属性是只读的，这样就不行了
    SocialSecurityNumber = ssn;
}
```

除非重新设计属性的值，否则就只能按照如下方式使用构造函数逻辑中的empSSN成员变量了：

```
public Employee(string name, int age, int id, float pay, string ssn)
{
...
  empSSN = ssn;
}
```

本节内容到此就结束了，请记住C#倾向于用属性来封装数据。这些句法实体的目的和传统的访问方法和修改方法相同。属性的好处是，对象的使用者可以使用单个命名的项来控制内部数据点。

源代码 EmployeeApp 项目的源代码位于Chapter 5子目录下。

5.7.5 静态属性

回想一下本章前面介绍的static关键字的作用，现在你应该知道使用C#属性语法的原因了，我们可以形式化静态属性。在StaticDataAndMembers项目中，SavingsAccount类有两个公共的静态属性，分别用于获取和设置利率。但更标准的做法是将该数据点封装在一个属性中。这样的话，即使没有分别获取和设置利率的方法，你也可以定义下面的类属性（注意static关键字的用法）：

```
//简单的储蓄账户类
class SavingsAccount
{
  //实例级别的数据
  public double currBalance;

  //静态数据点
  private static double currInterestRate = 0.04;

  //静态属性
  public static double InterestRate
  {
    get { return currInterestRate; }
    set { currInterestRate = value; }
  }
...
}
```

如果你想使用这个属性，而不用之前的两个静态方法，可以按照如下方式更新Main()方法：

```
//通过属性输出当前利率
Console.WriteLine("Interest Rate is: {0}", SavingsAccount.InterestRate);
```

5.8 自动属性

在创建属性封装数据的过程中，大多数C#属性都在set作用域中包含业务逻辑。但有时候你可能只需要简单的获取和设置值，而不需要任何实现逻辑。这意味你能够以如下所示的代码结尾：

```
// 使用标准属性语法的Car类型
class Car
```

```
{
  private string carName = "";
  public string PetName
  {
    get { return carName; }
    set { carName = value; }
  }
}
```

在这种情况下，多次定义私有返回字段和简单属性就显得十分累赘了。例如，你正在创建一个需要15个私有字段数据点的类，你编写了15个相关的属性，而它们仅仅对封装服务做了简单的包装。

要简化封装字段数据的过程，可以使用自动属性语法。顾名思义，该特性使用一种新的语法，为编译器减轻了定义私有返回字段和相关C#属性成员的工作。考虑下面对于Car类的改造，我们使用这种语法快速创建了3个属性：

```
class Car
{
  // 自动属性
  public string PetName { get; set; }
  public int Speed { get; set; }
  public string Color { get; set; }
}
```

说明 Visual Studio提供了prop代码段。输入prop并按下Tab键两次，IDE将为一个新的自动属性生成一段开始代码！你还可以在定义的各个部分使用Tab键来添加细节，赶紧动手试试吧！

在定义自动属性时，只指定访问修饰符、实际的数据类型、属性名称和空的get/set作用域。在编译时，编译器为类型自动生成了私有返回字段和适当的get/set实现逻辑。

说明 自动生成的私有返回字段的名称在你的C#代码库中是不可见的。查看它的唯一方法是使用ildasm.exe这样的工具。

但与传统的C#属性不同的是，不允许构建只读或只写的自动属性。也许你会认为像下面那样仅仅忽略属性声明中的get;或set;就可以了：

```
// 只读属性? 错误
public int MyReadOnlyProp { get; }

// 只写属性? 错误
public int MyWriteOnlyProp { set; }
```

而这将导致编译器错误。自动属性在定义时必须同时支持读写功能。但是，定义更严格的get或set是可以的：

```
// 自动属性必须同时可以读写
public string PetName { get; set; }
```

5.8.1 与自动属性交互

由于编译器在编译时才会定义私有的返回字段，所以定义自动属性的类通常都需要使用属性语法来获取和设置实际的值。注意到这一点是很重要的，因为很多程序员在类定义的内部直接使用私有字段，而这在这种情况下是行不通的。例如，如果Car类提供了DisplayStats()方法，实现该方法时需要使用属性名称：

```
class Car
{
  // 自动属性
  public string PetName { get; set; }
  public int Speed { get; set; }
  public string Color { get; set; }

  public void DisplayStats()
  {
    Console.WriteLine("Car Name: {0}", PetName);
    Console.WriteLine("Speed: {0}", Speed);
    Console.WriteLine("Color: {0}", Color);
  }
}
```

在使用定义了自动属性的对象时，可以用属性语法来设置和获取值：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Automatic Properties *****\n");

  Car c = new Car();
  c.PetName = "Frank";
  c.Speed = 55;
  c.Color = "Red";

  Console.WriteLine("Your car is named {0}?  That's odd...",
    c.PetName);
  c.DisplayStats();

  Console.ReadLine();
}
```

5.8.2 关于自动属性和默认值

你可以直接在代码库中使用封装了数字或布尔数据的自动属性，因为隐藏的返回字段将设置一个可以直接使用的安全的默认值。但如果自动属性包装了另一个类变量，隐藏的私有引用类型的默认值也将设置为null。

考虑下面新的Garage类，它使用了两个自动属性（当然真实的车库类会维护Car对象的集合，目前我们先忽略这一细节）：

```
class Garage
{
  // 隐藏的int返回字段设置为0
  public int NumberOfCars { get; set; }

  // 隐藏的Car返回字段设置为null
```

```
    public Car MyAuto { get; set; }
}
```

由于C#会为字段数据提供默认值，你能知道NumberOfCars的值（自动设置为0），但当你直接调用MyAuto时，将在运行时得到“空引用异常”。这是因为没有为后台使用的Car成员变量设置新的对象：

```
static void Main(string[] args)
{
  ...
  Garage g = new Garage();

  // 打印默认值0
  Console.WriteLine("Number of Cars: {0}", g.NumberOfCars);

  // 运行时错误！返回字段现在为null
  Console.WriteLine(g.MyAuto.PetName);
  Console.ReadLine();
}
```

由于私有的返回字段是在编译时创建的，所以你不能使用C#的字段初始化语法用new关键字直接分配引用类型。这项工作必须在类构造函数内部执行，以确保对象以安全的方式诞生。例如：

```
class Garage
{
  // 隐藏的返回字段设置为0
  public int NumberOfCars { get; set; }

  // 隐藏的返回字段设置为null
  public Car MyAuto { get; set; }

  // 必须使用构造函数重写分配给隐藏返回字段的默认值
  public Garage()
  {
    MyAuto = new Car();
    NumberOfCars = 1;
  }
  public Garage(Car car, int number)
  {
    MyAuto = car;
    NumberOfCars = number;
  }
}
```

更新之后，就可以像下面这样将Car对象传入Garage对象：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Automatic Properties *****\n");

  // 创建一个汽车
  Car c = new Car();
  c.PetName = "Frank";
  c.Speed = 55;
  c.Color = "Red";
  c.DisplayStats();

  // 将汽车放入车库中
  Garage g = new Garage();
  g.MyAuto = c;
  Console.WriteLine("Number of Cars in garage: {0}", g.NumberOfCars);
```

```
    Console.WriteLine("Your car is named: {0}", g.MyAuto.PetName);

    Console.ReadLine();
  }
```

这是C#编程语言中非常好的特性，你可以使用这种简单的语法在类中定义大量属性。当然，如果属性除了获取和设置实际的私有字段外，还需要额外的代码（如数据验证逻辑、编写事件日志、与数据库通信等），你就需要手工定义“普通的”.NET属性。C#自动属性只能对（编译器生成的）私有数据进行简单的封装。

源代码 AutoProps项目的源代码位于Chapter 5子目录下。

5.9 对象初始化语法

在本章前面的内容中，在新建对象时可以通过构造函数指定一些初始值。而属性允许我们安全地获取或设置实际的数据。当你使用其他人设计的类时，包括使用.NET基础类库中的类时，你往往会发现没有一个构造函数允许我们设置所有的状态数据。因此，程序员常常被迫去挑选最合适的构造函数，然后再使用所提供的一些属性赋值。

为了简化新建对象的过程，C#提供了*对象初始化语法*。使用这项技术，只用少量代码就可以创建对象并设置一些属性和公共字段。在语法上，对象初始化器的组成为：大括号内部用逗号分隔的指定值列表。初始化列表中的每个成员都映射为正在初始化的对象中公共字段或公共属性的名称。

为了查看该语法的实际情况，我们新建一个Console Application，取名为ObjectInitializers。现在，考虑下面的用自动属性创建的简单类`Point`（在本例中，使用自动属性并不是必需的，但它能使我们的代码更简洁）：

```
class Point
{
  public int X { get; set; }
  public int Y { get; set; }

  public Point(int xVal, int yVal)
  {
    X = xVal;
    Y = yVal;
  }
  public Point() { }

  public void DisplayStats()
  {
    Console.WriteLine("[{0}, {1}]", X, Y);
  }
}
```

现在，考虑下面的创建`Point`对象的方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Init Syntax *****\n");
```

```
    // 通过手动设置各个属性来创建Point
    Point firstPoint = new Point();
    firstPoint.X = 10;
    firstPoint.Y = 10;
    firstPoint.DisplayStats();

    // 或通过自定义构造函数创建Point
    Point anotherPoint = new Point(20, 20);
    anotherPoint.DisplayStats();

    // 或使用对象初始化语法创建Point
    Point finalPoint = new Point { X = 30, Y = 30 };
    finalPoint.DisplayStats();
    Console.ReadLine();
  }
```

最后一个Point变量没有像以前那样使用自定义构造函数，但仍然可以对公共的X、Y属性赋值。在后台调用的是类型的默认构造函数，然后再给指定的属性赋值。因此，对象初始化语法只是使用默认构造函数创建类变量并设置各个属性状态数据的语法的简写形式。

5

5.9.1 使用初始化语法调用自定义构造函数

前面示例中初始化的Point类型隐式地调用了类型的默认构造函数：

```
// 这里隐式地调用了默认构造函数
Point finalPoint = new Point { X = 30, Y = 30 };
```

如果你想看得更清楚一点，也可以显式地调用默认构造函数：

```
// 这里显式地调用了默认构造函数
Point finalPoint = new Point() { X = 30, Y = 30 };
```

要知道，在使用初始化语法构建类型时，可以调用类定义的任意构造函数。我们的Point类型当前定义了包含两个参数的构造函数，用来设置(X，Y)坐标。因此，下面Point声明的结果是X和Y的值都是100，而不是构造函数参数所指定的10和16：

```
// 调用自定义构造函数
Point pt = new Point(10, 16) { X = 100, Y = 100 };
```

对于当前Point类型的定义，使用初始化语法调用自定义构造函数除了增加一些冗余代码外没有任何好处。但是，如果Point类型提供了一个新的构造函数，允许调用者设置颜色（通过自定义的PointColor枚举），那么这种自定义构造函数和对象初始化语法的组合就显得十分清晰了。假设修改后的Point如下：

```
public enum PointColor
{ LightBlue, BloodRed, Gold }

class Point
{
  public int X { get; set; }
  public int Y { get; set; }
  public PointColor Color{ get; set; }

  public Point(int xVal, int yVal)
  {
    X = xVal;
```

```
    Y = yVal;
    Color = PointColor.Gold;
  }

  public Point(PointColor ptColor)
  {
    Color = ptColor;
  }

  public Point()
    : this(PointColor.BloodRed){ }

  public void DisplayStats()
  {
    Console.WriteLine("[{0}, {1}]", X, Y);
    Console.WriteLine("Point is {0}", Color);
  }
}
```

使用这个新的构造函数，可以创建一个金色的点（位置在(90, 20)）：

```
// 使用初始化语法调用自定义构造函数
Point goldPoint = new Point(PointColor.Gold){ X = 90, Y = 20 };
Console.WriteLine("Value of Point is: {0}", goldPoint.DisplayStats());
```

5.9.2 初始化内部类型

本章前面简单提到了（将在第6章中全面介绍）包含（has-a）关系允许我们通过定义已知类的成员变量来组合新类。例如，假设你有Rectangle类，这个类使用Point类型来表示左上角和右下角的坐标。由于自动属性将所有的内部类变量设置为null，所以你需要用“传统的”属性语法来实现这个新类：

```
class Rectangle
{
  private Point topLeft = new Point();
  private Point bottomRight = new Point();

  public Point TopLeft
  {
    get { return topLeft; }
    set { topLeft = value; }
  }
  public Point BottomRight
  {
    get { return bottomRight; }
    set { bottomRight = value; }
  }

  public void DisplayStats()
  {
    Console.WriteLine("[TopLeft: {0}, {1}, {2} BottomRight: {3}, {4}, {5}]",
      topLeft.X, topLeft.Y, topLeft.Color,
      bottomRight.X, bottomRight.Y, bottomRight.Color);
  }
}
```

使用对象初始化语法，可以像下面这样新建Rectangle变量并设置内部Point：

```
// 创建并初始化Rectangle
Rectangle myRect = new Rectangle
{
```

```
    TopLeft = new Point { X = 10, Y = 10 },
    BottomRight = new Point { X = 200, Y = 200}
  };
```

对象初始化语法的好处是，从根本上减少了键盘敲击的次数（假设没有合适的构造函数）。传统的建立Rectangle的方法如下：

```
// 旧的方法
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point();
p2.X = 200;
p2.Y = 200;
r.BottomRight = p2;
```

你可能会不太习惯这种对象初始化语法，但一旦习惯之后，你将对这种快速无扰的新建对象状态的方式感到十分欣慰。

在本章最后，我将介绍3个较小的话题——常量数据、只读字段和分部类定义，它们将使你更好地了解如何构建封装良好的类。

源代码 ObjectInitializers项目的源代码位于Chapter 5子目录下。

5.10 常量数据

C#提供了const关键字来定义常量（它在赋初始值后从未变过）。你可能会猜到，如果我们要为应用程序定义逻辑上和某个类或结构相关的一组已知值的话，就非常有用。

假设我们正在构建一个叫MyMathClass的工具类，并且需要定义一个PI值（假设是3.14）。先新建一个叫ConstData的控制台应用程序。如果不希望其他开发人员改变代码中的这个值，可以使用如下常量来定义PI：

```
namespace ConstData
{
  class MyMathClass
  {
    public const double PI = 3.14;
  }

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Fun with Const *****\n");
      Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
      // 错误！不能改变常量
      // MyMathClass.PI = 3.1444;

      Console.ReadLine();
    }
  }
}
```

注意，我们使用类名前缀（即MyMathClass.PI）引用MyMathClass定义的常量数据。这是因为类的常量字段是隐式静态的。然而，我们可以在类型成员中定义和访问局部常量。例如：

```
static void LocalConstStringVariable()
{
  // 局部常量可以被直接访问
  const string fixedStr = "Fixed string Data";
  Console.WriteLine(fixedStr);

  // 错误
  fixedStr = "This will not work!";
}
```

不管在哪里定义常量，有一点我们需要记住，定义常量时必须为常量指定初始值。因此，如果按如下代码修改MyMathClass，在类构造函数中为PI的值进行赋值：

```
class MyMathClass
{
  // 尝试在构造函数中设置PI
  public const double PI;

  public MyMathClass()
  {
  // 错误
    PI = 3.14;
  }
}
```

我们将会收到编译时错误。有这种限制是因为在编译时必须知道常量的值。我们知道，构造函数是在运行时调用的。

5.10.1　只读字段

和常量紧密联系的概念是只读字段数据（不要和只读属性混淆）。和常量相似，只读字段不能在赋初始值后改变。然而，和常量不同的是，赋给只读字段的值可以在运行时决定，因此在构造函数作用域中进行赋值是合法的（其他地方不行）。

如果直到运行时才知道字段值（可能我们需要读取外部文件来获得值），并且希望之后值不会被改变，那么上面的做法就很有用。假设按如下代码更新MyMathClass：

```
class MyMathClass
{
  // 可以在构造函数中为只读字段赋值，其他地方不行
  public readonly double PI;

  public MyMathClass ()
  {
    PI = 3.14;
  }
}
```

同样，在构造函数作用域之外用readonly为字段赋值的任何尝试都会导致编译器错误：

```
class MyMathClass
{
  public readonly double PI;
  public MyMathClass ()
```

```
    {
      PI = 3.14;
    }

    // 错误
    public void ChangePI()
    { PI = 3.14444;}
  }
```

5.10.2 静态只读字段

和常量字段不同，只读字段不是隐式静态的。因此，如果要从类级别公开PI，就必须显式使用static关键字。如果在编译时已经知道静态只读字段的值，那么它的初始赋值方式将和常量字段非常相似：


```
class MyMathClass
{
  public static readonly double PI = 3.14;
}

class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Const *****");
    Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
    Console.ReadLine();
  }
}
```

但是，如果直到运行时才知道静态只读字段的值，就必须使用本章前面描述过的静态构造函数。

```
class MyMathClass
{
  public static readonly double PI;

  static MyMathClass()
  { PI = 3.14; }
}
```

源代码 ConstData项目的源代码位于Chapter 5子目录下。

5.11 分部类型

最后，理解C#中partial关键字的作用也是很重要的。一个产品级别的类很容易就会达到成百上千行代码。而一个类常常定义在单个*.cs文件中，这就将导致代码文件非常长。而在创建类时，大多数代码在确立之后基本上就可以忽略不计。例如，字段数据、属性和构造函数在生产过程中很少变动，而方法却需要经常修改。

如果愿意，你可以将一个类分布到多个C#文件中，这样可以从样板代码中分离出有用的成员。为

了演示这一点，在Visual Studio中打开本章前面创建的EmployeeApp项目，打开Employee.cs文件。当前该文件包含的是该类所有部分的代码：

```
class Employee
{
  // 字段数据

  // 构造函数

  // 方法

  // 属性
}
```

使用分部类将构造函数和字段数据转移到全新的Employee.Internal.cs文件中（请注意，文件名可以随意起，我在这里使用Internal是想表示类的内部）。第一步是向当前的类定义中添加partial关键字，再剪切要转移到新文件中的代码：

```
// Employee.cs
partial class Employee
{
  // 方法

  // 属性
}
```

然后，假设已经在项目中插入了新的类文件，将数据字段和构造函数粘贴到新文件中。此外，还必须在类定义中添加partial关键字。例如：

```
// Employee.Internal.cs
partial class Employee
{
  // 字段数据

  // 构造函数
}
```

说明　记住！分部类的每个部分都必须用partial关键字标记。

编译这个修改后的项目，你会发现一切都没有变化。分部类的整个理念都是在设计时实现的。编译应用程序后，程序集中只存在唯一的类。定义分部类唯一的要求是类型名称（本例中为Employee）必须相同，并且必须定义在相同的.NET命名空间中。

实际上，你没有必要频繁地使用分部类。但Visual Studio却总是在后台使用它们。在本书后面开始使用WPF或ASP.NET等研究GUI应用程序开发时，你会发现Visual Studio将设计器生成的代码隔离到分部类中，而你只需要关注应用中特定的程序逻辑。

源代码　EmployeeAppPartial项目的源代码位于Chapter 5子目录下。

5.12　小结

本章的宗旨是介绍C#类类型的作用。我们看到，类可以接受任意构造函数来允许对象用户在创建对象时创建状态。本章还演示了几个类设计技术（以及相关关键字）。回忆一下，this关键字可以用于访问当前对象，static关键字可以用于在类（不是对象）级别定义字段和成员，而const关键字（以及readonly修饰符）可以用来定义在赋初始值之后永远不能改变的数据点。

本章大部分笔墨花在了OOP第一个支柱的细节：封装。在这里我们学习了C#的访问修饰符以及类型属性、对象初始化语法以及分部类的作用。有了这些，我们现在就可以转到下一章来学习使用继承和多态构建一组相关类。

第6章 继承和多态

第5章研究了OOP的第一个支柱——封装，并探讨了如何使用构造函数和各种成员（构造函数、字段、属性、方法、常量、只读字段等）来构建一个定义明确的类类型。本章会关注OOP其余两个支柱：继承和多态。

首先，我们会学习如何使用继承来构建一组相关类。我们会看到，这种形式的代码重用允许我们在父类中定义通用功能，并且这种功能可以被子类所使用（或者改变）。然后，我们会学习如何使用虚成员和抽象成员在类层次结构中创建多态接口。最后介绍.NET基础类库中超级父类`System.Object`的作用。

6.1 继承的基本机制

第5章介绍过，继承是OOP的一个方面，可以促进代码重用。更具体地说，代码重用归为两类：继承（“is-a”关系）和包含/委托模型（“has-a”关系）。让我们通过研究经典的“is-a”继承模型来开始本章。

在类之间创建“is-a”关系，也就在两个或两个以上类类型之间构建了依赖关系。经典继承的基本思想是新的类可以利用既有类的功能。让我们新建一个控制台应用程序BasicInheritance来开始非常简单的示例。假设我们已经设计了一个简单类`Car`来模拟汽车的基本细节：

```
// 简单的基类
class Car
{
  public readonly int maxSpeed;
  private int currSpeed;

  public Car(int max)
  {
    maxSpeed = max;
  }
  public Car()
  {
    maxSpeed = 55;
  }

  public int Speed
  {
    get { return currSpeed; }
    set
    {
```

```
        currSpeed = value;
        if (currSpeed > maxSpeed)
        {
          currSpeed = maxSpeed;
        }
      }
    }
  }
```

注意，Car类利用了封装服务，使用公共属性Speed来控制对currSpeed私有字段的访问。现在可以编写Car类型，如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");
  // 创建一个Car对象并设置最大速度
  Car myCar = new Car(80);

  // 设置当前速度并输出
  myCar.Speed = 50;
  Console.WriteLine("My car is going {0} MPH", myCar.Speed);
  Console.ReadLine();
}
```

6

6.1.1 指定既有类的父类

现在假设要构建一个新类MiniVan。和基本的Car相似，我们希望定义MiniVan类支持最大速度、当前速度以及允许对象用户修改对象状态的Speed属性。显然，Car和MiniVan类是关联的，其实我们可以说MiniVan“is-a”（是一个）Car。“is-a”关系（正式的术语是经典继承）允许我们构建新的类定义，扩展既有类的功能。

作为新类基础的既有类叫做基类或父类。基类的作用是为扩展它的类定义所有公共数据以及成员。扩展类叫做派生类或子类。在C#中，我们在类定义中利用冒号操作符在类之间创建“is-a”关系。假设你创建了下面的MiniVan类：

```
// MiniVan是一个Car
class MiniVan : Car
{
}
```

但该新类当前没有定义任何成员。那么，我们从Car基类扩展出的MiniVan获得了什么呢？简而言之，MiniVan对象现在拥有定义在父类中的每一个公共成员的访问权限。

说明 虽然构造函数一般定义为公共的，但派生类从来没有继承父类的构造函数。构造函数仅仅用于在内部定义的类。

有了两个类类型之间的这种关系，我们现在就可以这样使用MiniVan类了：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");
...
```

```
    // 创建MiniVan对象
    MiniVan myVan = new MiniVan();
    myVan.Speed = 10;
    Console.WriteLine("My van is going {0} MPH",
      myVan.Speed);
    Console.ReadLine();
  }
```

注意，尽管我们没有为MiniVan类增加任何成员，但是我们可以直接访问父类的公共属性Speed，因此就重用了代码。相比创建一个与Car有同样成员（如Speed属性）的MiniVan类来说，这种方法要好得多。如果两个类的代码完全一样，就需要维护两个代码体，这当然浪费时间。

一定要记住，继承保护了封装。因此下面的代码将导致编译器错误，不能通过对象引用来访问私有成员。

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");
...
  // 生成一个MiniVan对象
  MiniVan myVan = new MiniVan();
  myVan.Speed = 10;
  Console.WriteLine("My van is going {0} MPH",
    myVan.Speed);

  // 错误！不能访问私有成员
  myVan.currSpeed = 55;
  Console.ReadLine();
}
```

还有，如果MiniVan定义了自己的成员，它就不能访问Car基类的任何私有成员。此外，私有成员只能被定义它的类访问。例如，下面的MiniVan中的方法会导致编译器错误：

```
// MiniVan类派生自Car类
class MiniVan : Car
{
  public void TestMethod()
  {
    // 正确！可以在派生类型中访问父类的公共成员
    Speed = 10;

    // 错误！不能在派生类型中访问父类的私有成员
    currSpeed = 10;
  }
}
```

6.1.2　多个基类

说到基类，要记住很重要的一点：C#要求一个类只能有一个直接基类。我们不能创建直接派生自两个或两个以上基类的类类型［这项技术（诸如非托管C++等其他C系列的语言所支持的）叫做多重继承，简写为MI］。如果你打算创建一个能够指定两个直接父类的类，你就会收到编译器错误。如下代码所示：

```
// 非法！C#不允许类的多重继承
class WontWork
  : BaseClassOne, BaseClassTwo
{}
```

在第8章中我们会看到，.NET平台确实允许某一个类（或结构）类型实现许多独立的接口。这样，C#类型可以实现很多行为，同时又避免了多重继承引起的复杂性。还有，虽然一个类只能有一个直接基类，但是一个接口可以直接从多个接口派生。使用这个技巧，可以构建灵活的接口层次来建模复杂的行为（同样，见第8章）。

6.1.3 sealed关键字

C#提供了另外一个关键字sealed来防止发生继承。如果我们将类标记为sealed，编译器将不会允许我们从这个类型派生。例如，假设我们认为进一步扩展MiniVan类是无意义的：

```
// MiniVan类不会被扩展
sealed class MiniVan : Car
{
}
```


如果尝试从这个类派生的话，就会收到一个编译时错误：

```
// 错误！不能扩展用sealed关键字标记的类
class DeluxeMiniVan
  : MiniVan
{}
```

通常，设计一个工具类时对类进行密封很有意义。例如，System命名空间定义了很多密封类。可以打开Visual Studio 的对象浏览器（通过View菜单）并且选择定义在mscorlib.dll程序集中的System命名空间的String类来亲自验证。注意图6-1里Summary窗口中突出显示的sealed关键字的用法。

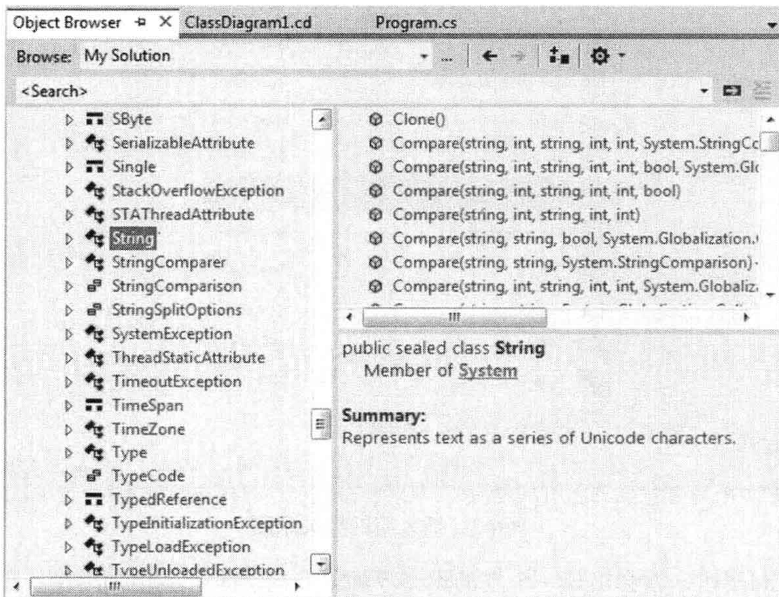


图6-1 基础类库定义了许多密封类型

因此，和MiniVan相似，如果尝试构建新的类来扩展System.String，将会收到一个编译时错误：

```
// 另一个错误！不能扩展已经被标注为sealed的类
class MyString
  : String
{}
```

说明　在第4章中我们已经知道，C#结构总是隐式密封的（见表4-3）。因此，我们永远不可以从结构继承结构，从类继承结构或从结构继承类。结构只能用于建模独立的、用户定义的数据类型。如果希望使用is-a关系，必须使用类。

你可能已经猜到了，在本章余下部分中还会介绍更多有关继承的细节。至此，只需记住冒号操作符用来创建基类和派生类的关系，而sealed关键字防止继承的发生。

6.2　回顾 Visual Studio 类关系图

第2章简单介绍过，Visual Studio 允许我们在设计时可视地创建基类/派生类关系。为了利用IDE的这个功能，第一步是在当前项目中包含新的类关系文件。通过访问Project→Add New Item菜单项，然后选择Class Diagram图标来实现（如图6-2所示，将文件名ClassDiagram1.cd重命名为Cars.cd）。

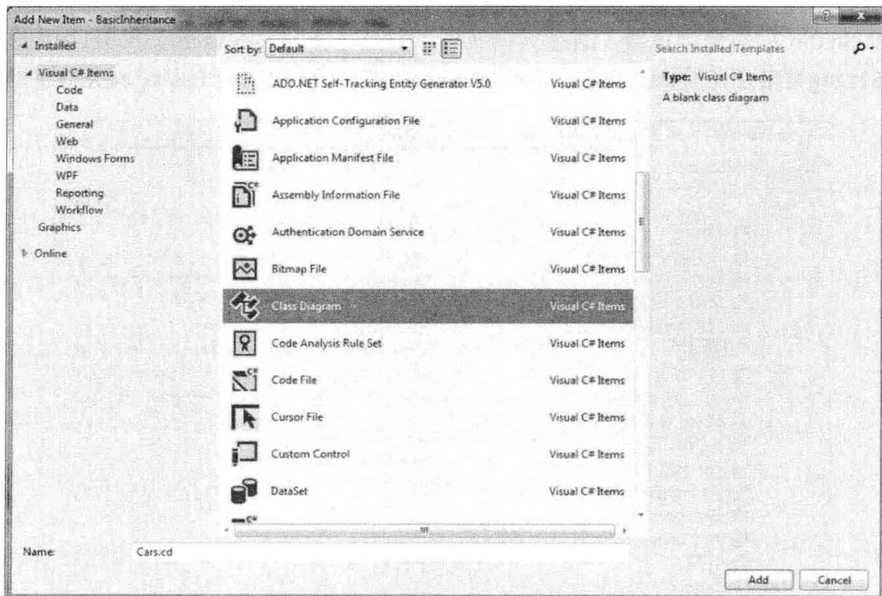


图6-2　插入新的类关系图

只要单击Add按钮，就会出现一个空的设计界面。要向类设计器增加类型，只需要从Solution Explorer窗口将每个文件拖到界面上即可。要知道，从可视化设计器上删除一个项时（选中它并按Del键），不会同时删除关联的源代码，而只是将它从设计器界面上移除。当前类的层次结构如图6-3所示。

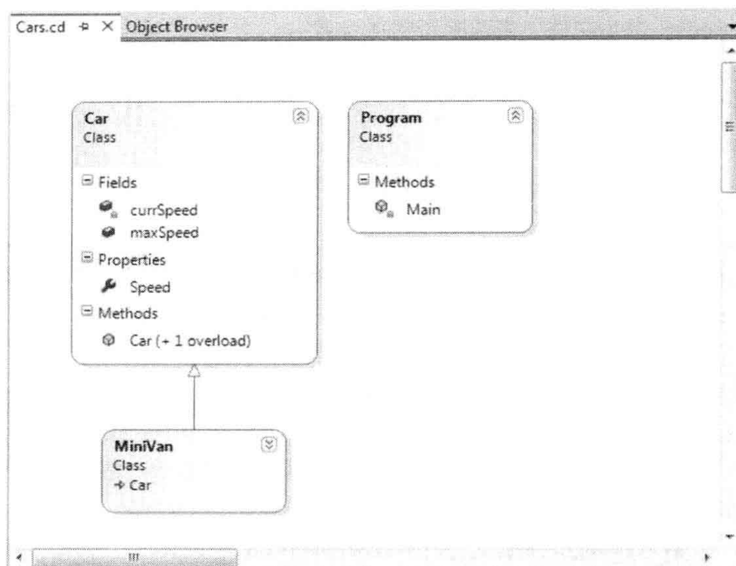


图6-3 Visual Studio的可视化设计器

说明 作为一种快捷方式，如果希望将项目中所有当前类型都添加到设计器界面，只需要选择Solution Explorer中的Project节点，然后单击Solution Explorer窗口右上角的View Class Diagram（查看类关系图）按钮。

除了显示当前应用程序中类型的关系之外，第2章中我们还使用Class Designer Toolbox和Class Details窗口创建过全新的类型并填充它们的成员。

如果你希望在本书其他部分使用这些可视化工具，请尽管用。然而，请务必分析生成的这些代码，以便对这些工具帮我们做的事情有足够的了解。

源代码 BasicInheritance项目的源代码位于Chapter 6子目录下。

6.3 OOP 的第二个支柱：继承

既然已经知道了继承的基本语法，就让我们创建更复杂的示例来了解构建类继承的更多细节吧。我们会重用第5章中设计的Employee类。首先，创建一个全新的C#控制台应用程序Employees。

然后，单击Project→Add Existing Item菜单项，再转到Employee.cs和Employee.Internals.cs文件的位置。选择每一个项（按住Ctrl键单击），然后单击Add按钮。Visual Studio会将每一个文件复制到当前项目。

在构建派生类之前，还有一个细节需要注意。由于Employee在EmployeeApp项目中被创建，这个类就被同名的.NET命名空间包装了。第14章会研究命名空间的细节，然而为了简单，我们将当前命名

空间重命名为Employees（两个文件）来匹配新的项目名：

```
// 确保改变两个C#文件中的命名空间名
namespace Employees
{
  partial class Employee
  {...}
}
```

说明　为了全面地检查程序，需要编译程序并按Ctrl+F5组合键运行项目。尽管这时该程序不会做任何事情，但这可以验证是否会产生编译器错误。

我们的目标是创建一组类来建模公司中的各种员工类型。假设我们希望利用Employee类的功能创建两个新类（SalesPerson和Manager）。我们可以看到SalesPerson是一个Employee（Manager也一样）。在经典的继承模型中，基类（如Employee）用于定义所有派生类型共有的一般特征。子类（如SalesPerson和Manager）通过增加特定的行为来扩展这些一般功能。

对于我们的示例，假设Manager类通过记录股票期权的数量来扩展Employee，而SalesPerson类提供了销售量。使用下面的自动属性插入定义了Manager类的新的类文件（Manager.cs）：

```
// 经理需要知道它们的股票期权数量
class Manager : Employee
{
  public int StockOptions { get; set; }
}
```

然后，使用一个合适的自动属性增加一个新的类文件（SalesPerson.cs）来定义SalesPerson类：

```
// 销售人员需要知道它们的销售量
class SalesPerson : Employee
{
  public int SalesNumber { get; set; }
}
```

现在已经建立了一种“is-a”关系，SalesPerson和Manager自动继承了Employee基类的所有公共成员。例如，按如下所示更新Main()方法：

```
// 创建子类对象并访问基类的功能
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");
  SalesPerson danny = new SalesPerson();
  danny.Age = 31;
  danny.Name = "Danny";
  danny.SalesNumber = 50;
  Console.ReadLine();
}
```

6.3.1　使用base关键字控制基类的创建

现在，只能使用免费的默认构造函数来创建SalesPerson和Manager（见第5章）。因此，假设为Manager类型新增了有6个参数的构造函数，按如下所示进行调用：

```
static void Main(string[] args)
{
...
  // 假设Manager有匹配这种签名的构造函数
  // (string fullName, int age, int empID,
  //  float currPay, string ssn, int numbOfOpts)
  Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
  Console.ReadLine();
}
```

从参数列表可以发现，大多数参数应该保存在由Employee基类定义的成员变量中。如果要这样做，就需要实现Manager类的自定义构造函数：

```
public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbOfOpts)
{
  // 这个属性由Manager类来定义
  StockOptions = numbOfOpts;

  // 使用从父类继承的属性来为传入参数赋值
  ID = empID;
  Age = age;
  Name = fullName;
  Pay = currPay;

  // 噢！这会导致编译器错误，因为SSN属性是只读的
  SocialSecurityNumber = ssn;
}
```

这种方法的第一个问题是，如果在父类中定义的属性（如SocialSecurityNumber属性）是只读的，就不能将传入的string参数赋给这个字段，正如这个自定义构造函数最后的代码语句所示。

第二个问题是，我们间接地创建了一个非常低效的构造函数。在C#下，一般基类的默认构造函数会在派生类构造函数执行之前被自动调用。之后，当前的实现会访问Employee基类的许多公共属性来创建其状态。因此，我们实际上在创建Manager对象时访问了公共属性7次（5个继承的属性以及两个构造函数调用）。

为了帮助优化派生类的创建，最好实现子类构造函数来显式调用合适的自定义基类构造函数而不是默认构造函数。这样，我们就可以减少对继承的初始化成员的调用次数（也就节省了处理时间）。现在使用base关键字来改进Manager类型的自定义构造函数：

```
public Manager(string fullName, int age, int empID,
               float currPay, string ssn, int numbOfOpts)
  : base(fullName, age, empID, currPay, ssn)
{
  // 这个属性由Manager类定义
  StockOptions = numbOfOpts;
}
```

在这里，base关键字挂接构造函数签名（和使用this关键字在单个类中串连构造函数的语法相似，见第5章），这代表派生构造函数将数据传递到最近的父构造函数中。在这里，我们显式调用由Employee定义的5个参数的构造函数，从而节省子类创建过程中不必要的调用。自定义的SalesPerson构造函数应该如下：

```
// 一般来说，所有子类应该显式调用合适的基类构造函数
public SalesPerson(string fullName, int age, int empID,
```

```
    float currPay, string ssn, int numbOfSales)
    : base(fullName, age, empID, currPay, ssn)
  {
    // 这个由自己处理
    SalesNumber = numbOfSales;
  }
```

说明 只要子类想访问由父类定义的公共或受保护的成员，我们就可以使用base关键字。不是只有构造函数逻辑才能使用这个关键字。在本章之后研究多态的时候，我们会看到这样用base关键字的很多示例。

最后，还记得吗，只要我们为类定义增加了自定义构造函数，默认构造函数就自动移除了。因此，要确保为SalesPerson和Manager类型重新定义默认构造函数，例如：

```
// 也在Manager类中添加默认构造函数
public SalesPerson() {}
```

6.3.2 家族的秘密：protected关键字

我们已经知道，公共项可以在任何地方直接访问，而私有项除了定义它的类之外，不能从其他对象进行访问。第5章介绍过，作为主流的面向对象语言，C#提供了另外一个定义成员可访问性的关键字：protected。

当基类定义了受保护数据或受保护成员时，它就创建了一组可以直接被任何后代访问的项。如果希望SalesPerson和Manager子类可以直接访问由Employee定义的数据区，可以按如下代码更新原来的Employee类定义：

```
// 受保护的状态数据
partial class Employee
{
  // 派生类现在可以直接访问这些信息了
  protected string empName;
  protected int empID;
  protected float currPay;
  protected int empAge;
  protected string empSSN;
  protected static string companyName;
...
}
```

在基类中定义受保护成员的好处在于：派生类型不再需要使用公共方法或属性来间接访问数据了。当然，可能的坏处在于：如果派生类型有权直接访问其父类内部数据，有可能会偶然绕过公共属性内设置的既有业务规则。当定义受保护成员时，也就创建了一种父类和子类之间的信任级别，编译器不会捕获任何违背类型业务规则的异常。

最后要理解，对对象用户来说，受保护数据可以认为是私有的（因为用户处于家族“之外”）。因此，下面的语句是不合法的：

```
static void Main(string[] args)
{
  // 错误！不能从对象实例中直接访问受保护数据
```

```
    Employee emp = new Employee();
    emp.empName = "Fred";
  }
```

说明 尽管protected字段数据会打破封装，但是定义protected方法是非常安全（和有用）的。在构建类层次结构时，我们通常会定义一组只能被派生类型使用的方法。

6.3.3 增加密封类

回忆一下，密封类是不能被其他类扩展的。我们提到过，这项技术通常用于设计工具类。然而，在构建类层次结构的时候，我们可能会发现某个分支应该被阻断，因为再进一步扩展，这个体系就会没有意义。例如，假设为程序增加了另一个类（PTSalesPerson）来扩展既有的SalesPerson类型。图6-4显示了当前的更新。

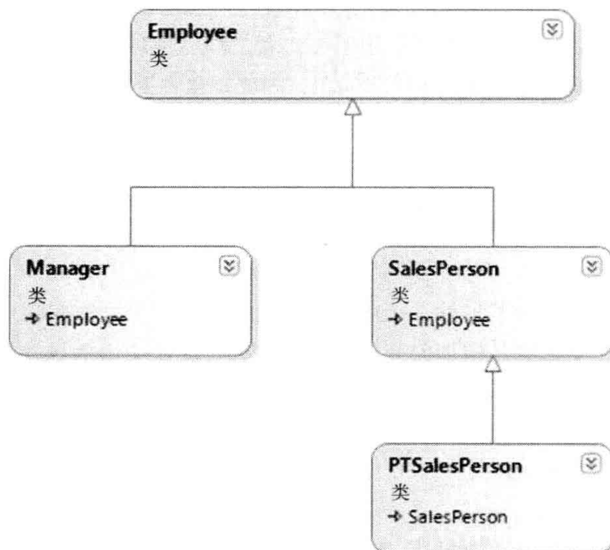


图6-4 PTSalesPerson类

PTSalesPerson类（当然）表示兼职销售员。为了说明清楚，假设希望确保没有其他开发人员可以从PTSalesPerson创建子类。（毕竟，兼职还能怎么分呢？）同样，为了防止其他人扩展类，可以使用sealed关键字：

```
sealed class PTSalesPerson : SalesPerson
{
  public PTSalesPerson(string fullName, int age, int empID,
                       float currPay, string ssn, int numbOfSales)
    :base (fullName, age, empID, currPay, ssn, numbOfSales)
  {
  }
    // 假设其他成员……
}
```

6.4　包含/委托编程

本章前面说过，代码重用分两类。我们刚研究过经典的“is-a”关系。在讨论OOP的第3个支柱（多态）之前，让我们先学习“has-a”关系（也叫做包含/委托模型或聚合）。首先新建一个类来建模员工的保险金。

```
// 这个类型会作为包含类
class BenefitPackage
{
  // 假设还有其他成员来表示牙齿、健康保险金等
  public double ComputePayDeduction()
  {
    return 125.0;
  }
}
```

很显然，在BenefitPackage类和员工类型之间建立“is-a”关系有点古怪。（Employee是一个BenefitPackage？我可不这么认为。）然而，很明显二者之间有某种关系。简单来说，我们希望表达每一个员工有（“has-a”）BenefitPackage这样的概念。为此，按如下所示更新Employee类的定义：

```
// 员工现在有保险金
partial class Employee
{
  // 包含BenefitPackage对象
  protected BenefitPackage empBenefits = new BenefitPackage();
...
}
```

至此，我们已经成功地包含了另外一个对象。然而，如果要公开被包含对象的功能给外部世界，就需要委托。简单地说，委托就是增加公共成员到包含类，以便使用被包含对象的功能。

例如，我们可以通过更新Employee类使用一个自定义属性来公开包含的empBenefits对象，并使用一个新方法GetBenefitCost()来使用它的内部功能：

```
public partial class Employee
{
  // 包含一个BenefitPackage对象
  protected BenefitPackage empBenefits = new BenefitPackage();

  // 公开对象的保险金行为
  public double GetBenefitCost()
  { return empBenefits.ComputePayDeduction(); }

  // 通过自定义属性公开对象
  public BenefitPackage Benefits
  {
    get { return empBenefits; }
    set { empBenefits = value; }
  }
...
}
```

在如下更新过的Main()方法中，注意如何和Employee类型定义的内部BenefitPackage类型进行交互：

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");
  ...
  Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
  double cost = chucky.GetBenefitCost();
  Console.ReadLine();
}
```

嵌套类型定义

第5章简单提到了嵌套类型的概念，这是刚才提到的"has-a"关系的另一种说法。在C#（以及其他.NET语言）中，我们可以直接在类或结构作用域中定义类型（枚举、类、接口、结构或委托）。如果这么做的话，被嵌套（或内部）的类型被认为是嵌套（或外部）类的成员，并且从运行时角度来说，可以像操作其他成员（字段、属性、方法和事件等）一样来操作嵌套类型。嵌套类型的语法非常简单明了：

```
public class OuterClass
{
  // 公共嵌套类型可以被任何人使用
  public class PublicInnerClass {}

  // 私有嵌套类型只可以被包含类的成员使用
  private class PrivateInnerClass {}
}
```

尽管语法很简洁，但是要理解为什么这么做却不那么简单。要理解这项技术，考虑如下嵌套类型的特征。

- 通过嵌套类型可以完全控制内部类型的访问级别，也就是可以声明为私有的（回忆一下，非嵌套类不能使用private关键字来声明）。
- 由于嵌套类型是包含类的成员，所以它可以访问包含类的私有成员。
- 通常，嵌套类型只用做外部类的辅助方法，而不是为外部世界所准备的。

如果类型嵌套了另一个类类型，它就可以创建这个类型的成员变量，就好像是数据点一样。然而，如果要从包含类型的外部使用嵌套类型，就必须限定嵌套类型的作用域。考虑如下代码：

```
static void Main(string[] args)
{
  // 创建并使用公共的内部类
  OuterClass.PublicInnerClass inner;
  inner = new OuterClass.PublicInnerClass();

  // 编译器错误！不能访问私有类
  OuterClass.PrivateInnerClass inner2;
  inner2 = new OuterClass.PrivateInnerClass();
}
```

要在我们的Employee示例中使用这个概念，可以假设现在在Employee类类型中直接嵌套类BenefitPackage：

```
partial class Employee
{
  public class BenefitPackage
  {
    // 假设有其他成员表示牙齿、健康保险金等
    public double ComputePayDeduction()
```

6

```
    {
      return 125.0;
    }
  }
...
}
```

嵌套的“深度”可以按需设定。例如，假设希望创建一个名为BenefitPackageLevel的枚举，它列举了员工可以选择的各种保险金级别。为了通过编程强制Employee、BenefitPackage和BenefitPackageLevel之间的紧密联系，我们可以按如下所示嵌套枚举：

```
// Employee嵌套BenefitPackage
public partial class Employee
{
  // BenefitPackage嵌套BenefitPackageLevel
  public class BenefitPackage
  {
    public enum BenefitPackageLevel
    {
      Standard, Gold, Platinum
    }

    public double ComputePayDeduction()
    {
      return 125.0;
    }
  }
...
}
```

在这种嵌套关系里需要注意如何使用枚举：

```
static void Main(string[] args)
{
...
  // 定义福利等级
  Employee.BenefitPackage.BenefitPackageLevel myBenefitLevel =
    Employee.BenefitPackage.BenefitPackageLevel.Platinum;
  Console.ReadLine();
}
```

太好了！至此我们已经研究了允许使用经典继承、包含和嵌套类型构建相关类型层次关系的许多关键字。如果还有一些细节不是很清楚，不要紧。我们会在本书其他部分构建许多其他层次关系。接着，让我们研究OOP的最后一个支柱：多态。

6.5　OOP 的第三个支柱：C#的多态支持

还记得吗，Employee基类定义了一个叫GiveBonus()的方法，它的原始实现如下：

```
public partial class Employee
{
  public void GiveBonus(float amount)
  {
    currPay += amount;
  }
...
}
```

因为这个方法使用public关键字来定义，我们现在可以给销售人员和经理（以及兼职销售人员）奖金了：

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");

  // 给每一个员工奖金
  Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
  chucky.GiveBonus(300);
  chucky.DisplayStats();
  Console.WriteLine();

  SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
  fran.GiveBonus(200);
  fran.DisplayStats();
  Console.ReadLine();
}
```

当前设计中的问题是，继承的GiveBonus()方法对所有子类都进行相同的操作。理想情况下，销售人员或兼职销售员的奖金应该和销售情况有关。可能经理应该得到额外的股票期权和加薪。因此，我们现在面临一个有趣的问题：相关类型如何对相同的请求做出不同的响应？

6.5.1 virtual和override关键字

多态为子类提供了一种方式，使其可以定义由其基类定义的方法，这种过程叫做*方法重写*。为了修改当前的设计，我们需要理解virtual和override关键字的作用。如果基类希望定义*可以*（不是必需的）由子类重写的方法，就必须用virtual关键字标志方法：

```
partial class Employee
{
  // 这个方法现在可以由派生类"重写"
  public virtual void GiveBonus(float amount)
  {
    Pay += amount;
  }
...
}
```

说明 用virtual关键字标记的方法（理所当然）称为虚方法。

如果子类希望改变虚方法的实现细节，就必须使用override关键字。例如，SalesPerson和Manager可以按如下所示重写GiveBonus()（假设PTSalesPerson不会重写GiveBonus()，因此也就只会继承由SalesPerson定义的版本）：

```
class SalesPerson : Employee
{
...
  // 销售人员的奖金受销售量的影响
  public override void GiveBonus(float amount)
  {
    int salesBonus = 0;
```

```
    if (numberOfSales >= 0 && numberOfSales <= 100)
      salesBonus = 10;
    else
    {
      if (numberOfSales >= 101 && numberOfSales <= 200)
        salesBonus = 15;
      else
        salesBonus = 20;
    }
    base.GiveBonus(amount * salesBonus);
  }
}

class Manager : Employee
{
...
  public override void GiveBonus(float amount)
  {
    base.GiveBonus(amount);
    Random r = new Random();
    numberOfOptions += r.Next(500);
  }
}
```

注意，重写方法完全可以通过base关键字来自由地使用默认行为。这样，我们就不需要完全重新实现GiveBonus()的逻辑，而是可以重用（或扩展）父类的默认行为。

假设Employee类当前的DisplayStats()方法被声明为虚的。这样，每一个子类都可以重写这个方法来说明销售量（对销售人员）或当前股票期权数量（对经理）。例如，如下所示的Manager版本的DisplayStats()方法（SalesPerson类也会以相似方式实现DisplayStats()）：

```
public override void DisplayStats()
{
  base.DisplayStats();
  Console.WriteLine("Number of Stock Options: {0}", StockOptions);
}
```

现在每一个子类都可以说明这些虚方法对自己意味着什么，每一个对象实例表现为更独立的实体：

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");

  // 更好的奖金系统
  Manager chucky = new Manager("Chucky", 50, 92, 100000, "333-23-2322", 9000);
  chucky.GiveBonus(300);
  chucky.DisplayStats();
  Console.WriteLine();

  SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000, "932-32-3232", 31);
  fran.GiveBonus(200);
  fran.DisplayStats();
  Console.ReadLine();
}
```

下面给出了到目前为止应用程序可能的测试运行结果：

```
***** The Employee Class Hierarchy *****

Name: Chucky
ID: 92
Age: 50
Pay: 100300
SSN: 333-23-2322
Number of Stock Options: 9337

Name: Fran
ID: 93
Age: 43
Pay: 5000
SSN: 932-32-3232
Number of Sales: 31
```

6.5.2　使用Visual Studio IDE重写虚方法

你可能已经知道，重写一个成员时必须记得每一个参数的类型，更不要说方法名和参数传递规则（`ref`、`out`和`params`等）。Visual Studio有一个非常有用的、可以在重写虚成员时使用的特性。如果在类类型作用域中输入“override”单词（然后按空格键），智能感知会自动显示定义在父类中的所有可重写成员的列表，如图6-5所示。

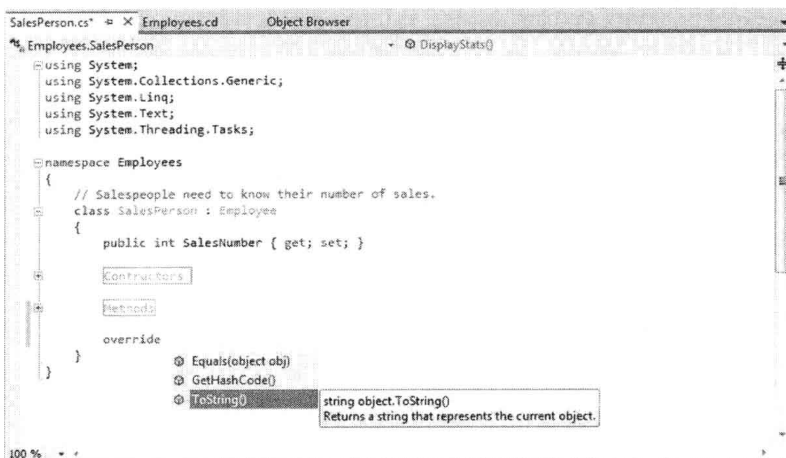


图6-5　在Visual Studio中快速查看可重写方法

如果我们选择一个成员并且按Enter，IDE会自动为我们填写方法存根。注意，我们还获得了一个调用父类版本虚成员的语句（如果不需要，完全可以删除这行）。例如，如果在重写`DisplayStats()`方法时使用这一技术，可能会看到下面自动生成的代码：

```
public override void DisplayStats()
{
  base.DisplayStats();
}
```

6.5.3 密封虚成员

回忆一下sealed关键字，它可以用于类类型来防止其他类型通过继承扩展其行为。你可能还记得，我们密封了PTSalesPerson，因为我们认为其他开发人员扩展这条继承线是无意义的。

补充一下，有时我们不希望密封整个类，而只希望防止派生类型来重写某个虚方法。例如，假设我们不希望兼职销售人员获得自定义的奖金，就要防止PTSalesPerson类重写GiveBonus()虚方法，我们只需要密封SalesPerson中的这个方法：

```
// SalesPerson密封了GiveBonus()方法
class SalesPerson : Employee
{
...
  public override sealed void GiveBonus(float amount)
  {
     ...
  }
}
```

在这里，SalesPerson确实重写了定义在Employee类中的GiveBonus()虚方法，然而它显式标记为密封的。因此，如果按如下所示的代码尝试在PTSalesPerson类中重写这个方法，你就会收到编译时错误：

```
sealed class PTSalesPerson : SalesPerson
{
  public PTSalesPerson(string fullName, int age, int empID,
                       float currPay, string ssn, int numbOfSales)
   :base (fullName, age, empID, currPay, ssn, numbOfSales)
  {
  }

  // 编译器错误！不能在PTSalesPerson类中重写这个方法，因为它是被密封的
  public override void GiveBonus(float amount)
  {
  }
}
```

6.5.4 抽象类

现在，Employee基类已经为它的后代提供了受保护的成员变量和两个可以被某个后代重写的虚方法（GiveBonus()和DisplayStats()）。虽然这不错，但是当前设计还有一个古怪的地方，我们可以直接创建Employee基类的实例：

```
// 这算什么意思
Employee X = new Employee();
```

在这个示例中，Employee基类的作用是为所有子类定义公共成员。在任何情况下，我们都不希望任何人直接创建这个类的实例，因为Employee类型本身是一个非常普通的概念。例如，如果我向你走过来，然后说："我是一个员工！"你可能非常想问我："你是什么样的员工？"顾问、培训师、行政助理、文字编辑或白宫助手等？

由于很多基类都是比较模糊的实体，好的设计师会防止在代码中直接创建新的Employee对象。在C#中，我们可以使用abstract关键字来强制这种编程方式，因此创建一个抽象基类：

```
// 将Employee类更新为抽象类来防止直接实例化
abstract partial class Employee
{
  ...
}
```

这样，如果现在尝试创建Employee类的实例，就会收到一个编译时错误：

```
// 错误！不能创建抽象类的实例
Employee X = new Employee();
```

定义了类却不能直接创建它，这乍看上去似乎很怪异。但要记住基类（抽象或非抽象）是非常有用的，它们包含了派生类型中所有的通用数据和功能。使用这种抽象，我们完全可以为那些不是具体实体的东西进行建模，如员工的“想法”（idea）。同样要了解的是，尽管我们不能直接创建抽象类，但在创建其派生类时，仍然会在内存中对其进行组装。因此对抽象类来说，定义若干在分配派生类时间接调用的构造函数是很正常（且常见）的。

至此，我们已经构建了一个相当有趣的员工层次结构。本章后面讨论C#强制转换规则时，我们会增加更多功能到这个应用程序中。图6-6显示了当前类型的核心设计。

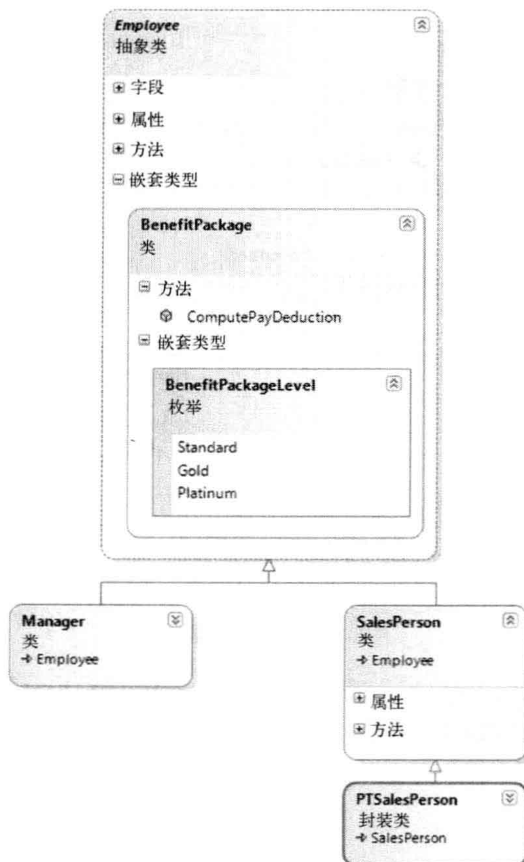


图6-6 雇员层次结构

源代码　Employees项目的源代码位于Chapter 6子目录下。

6.5.5　构建多态接口

如果类被定义为抽象基类（通过abstract关键字），它就可以定义许多抽象成员。当你希望定义没有提供默认实现而必须在每个派生类中实现的成员时，可以使用抽象成员。这样做就强制了每一个后代具有多态接口，它们需要自己处理抽象方法的细节。

简而言之，抽象基类的多态接口只是指一组虚的或者抽象的方法。其有趣之处不是一眼就可以看出的，因为OOP的这个特性可用来构建可高度扩展的灵活的应用程序。为了演示，我们会实现（以及一点小修改）第5章概览OOP支柱时粗略研究过的图形层次结构。首先，新建一个名为Shapes的C#控制台应用程序。

在图6-7中，注意Hexagon和Circle类型都扩展了Shape基类。和任何其他基类相似，Shape定义了许多所有后代都共有的成员（这里是PetName属性和Draw()方法）。

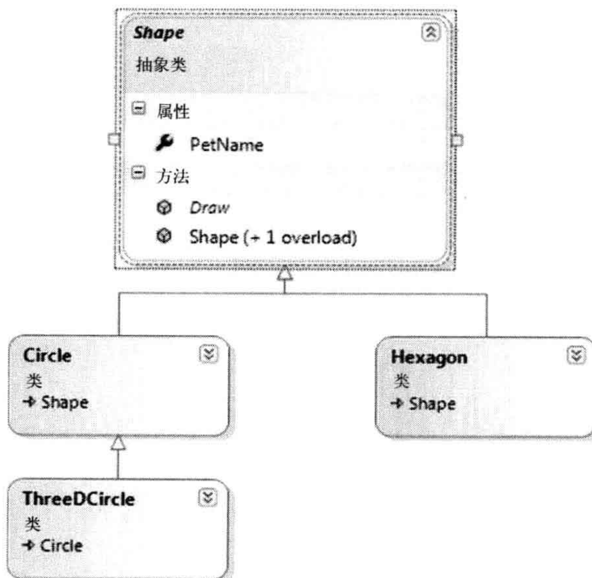


图6-7　图形层次结构

和员工层次结构相似，我们应该不希望允许对象用户直接创建Shape的实例，因为这个概念过于抽象。同样，为了防止直接创建Shape类型，可以把它定义为抽象类。同样，由于我们希望派生类型有自己独特的Draw()方法，那么我们就把它标记为virtual，并定义一个默认的实现。

```
// 层次结构中的抽象基类
abstract class Shape
{
  public Shape(string name = "NoName")
  { PetName = name; }
```

```
  public string PetName { get; set; }

  // 一个虚方法
  public virtual void Draw()
  {
    Console.WriteLine("Inside Shape.Draw()");
  }
}
```

注意Draw()虚方法提供了默认实现，仅仅是输出一个消息，通知我们调用的是Shape基类中的Draw()方法。现在回忆一下，如果一个方法标记为virtual，这个方法提供的默认实现会被所有派生类型自动继承。如果子类愿意这样继承，可以重写这个方法，但这不是必需的。因此，考虑如下Circle和Hexagon类型的实现：

```
// Circle没有重写Draw()
class Circle : Shape
{
  public Circle() {}
  public Circle(string name) : base(name){}
}

// Hexagon重写了Draw()
class Hexagon : Shape
{
  public Hexagon() {}
  public Hexagon(string name) : base(name){}
  public override void Draw()
  {
    Console.WriteLine("Drawing {0} the Hexagon", PetName);
  }
}
```

还记得吧，子类重写虚方法不是必需的（正如Circle的例子），这样抽象方法的作用就很明显了。因此，如果我们创建Hexagon和Circle类型的实例，会发现Hexagon知道如何正确地“绘制”自身（至少将正确的消息输出到控制台）。而Circle就有一点令人困惑：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Polymorphism *****\n");

  Hexagon hex = new Hexagon("Beth");
  hex.Draw();

  Circle cir = new Circle("Cindy");
  // 调用基类实现
  cir.Draw();
  Console.ReadLine();
}
```

现在看一下前面的Main()方法的输出结果：

```
***** Fun with Polymorphism *****

Drawing Beth the Hexagon
Inside Shape.Draw()
```

6

很明显，对于当前的层次结构而言，这不是非常明智的设置。要强制每一个子类重写Draw()方法，我们可以把Draw()定义为Shape类的一个抽象方法，也就是说我们没有提供默认的实现。在C#中可以使用abstract关键字将方法标记为抽象的。注意，abstract成员没有提供任何实现：

```
abstract class Shape
{
  // 强制所有子类来定义如何被呈现
  public abstract void Draw();
  ...
}
```

说明　抽象方法只可以定义在抽象类中。如果不是这样的话，就会收到编译器错误。

标记为abstract的方法是纯粹的协议。它们只是定义了名字、返回值（如果需要的话）和参数集合（如果需要的话）。在这里，抽象的Shape类通知了派生类型："我有一个无参方法Draw()。如果要从我派生的话，请填充细节。"

因此，我们就必须在Circle类中重写Draw()方法。如果不这样的话，Circle就应该是不可创建的抽象类型，必须使用abstract关键字来修饰（对于这个示例，很明显不适合）。这里是更新后的代码：

```
// 如果不实现抽象的Draw()方法，Circle也必须是抽象的，我们必须将其标记为abstract
class Circle : Shape
{
  public Circle() {}
  public Circle(string name) : base(name) {}
  public override void Draw()
  {
    Console.WriteLine("Drawing {0} the Circle", PetName);
  }
}
```

现在我们就能认为任何从Shape派生的类型都会有自己版本的Draw()方法。为了了解完整的多态，考虑如下的代码：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Polymorphism *****\n");

  // 创建一个图形对象数组
  Shape[] myShapes = {new Hexagon(), new Circle(), new Hexagon("Mick"),
    new Circle("Beth"), new Hexagon("Linda")};

  // 循环每一个项来和多态接口进行交互
  foreach (Shape s in myShapes)
  {
    s.Draw();
  }
  Console.ReadLine();
}
```

下面显示了修改的Main()方法的输出结果：

```
***** Fun with Polymorphism *****

Drawing NoName the Hexagon
Drawing NoName the Circle
Drawing Mick the Hexagon
Drawing Beth the Circle
Drawing Linda the Hexagon
```

Main()方法说明了多态的作用。尽管不能直接创建抽象基类的实例（Shape），我们完全可以使用抽象基类变量来保存指向任何子类的引用。因此，创建myShapes数组时，数组可以保存任何从Shape基类派生的对象（如果试图把非Shape兼容的对象放入数组，就会收到编译器错误）。

因为myShapes数组中的所有项都派生自Shape，我们知道它们都支持相同的多态接口（或者更直接地说，它们都有Draw()方法）。当我们在运行时迭代Shapes引用数组的时候，就已经决定了其实际类型。这样的话，就会调用正确版本的Draw()方法。

这个技术使得我们扩展当前的层次结构很安全。例如，假设我们从Shape抽象基类派生5个类（Triangle、Square等）。由于多态接口的使用，foreach循环中的代码不需要任何改动，因为编译器会强制只有Shape兼容的类型才能放到myShapes数组中。

6

6.5.6 成员投影

C#提供了逻辑上和方法重写相对的功能，叫做投影（shadowing）。正式地说，如果派生类定义的成员和定义在基类中的成员一致，派生类就投影了父类的版本。在真实情况下，如果我们（或者我们的团队）从一个不是自己创建的类来创建子类就很可能会发生这样的情况（例如，如果你购买第三方.NET软件包）。

举例说明，假设我们的同事或同学定义了一个叫ThreeDCircle的类，它有一个叫Draw()的无参方法：

```
class ThreeDCircle
{
  public void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

我们发现ThreeDCircle是（“is-a”）Circle，因此我们从既有的Circle类型进行派生：

```
class ThreeDCircle : Circle
{
  public void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

重编译之后，我们发现如下警告信息：

```
'Shapes.ThreeDCircle.Draw()' hides inherited member 'Shapes.Circle.Draw()'.
To make the current member override that implementation, add the override keyword.
Otherwise add the new keyword.
```

问题在于派生类（ThreeDCircle）包含了与继承的方法同名的方法。为了解决这个问题，我们有两个选择。第一种，我们可以只使用override关键字更新父版本的Draw()（就如编译器建议的那样）。这样，ThreeDCircle类型就可以按需扩展父类的默认行为。然而，如果我们对定义基类的代码没有访问权限（同样，比如在很多第三方类库中），我们就不能将Draw()方法修改为虚方法，因为我们不能访问代码文件！

第二种选择是，我们可以为派生类型（这里是ThreeDCircle）的Draw()成员添加new关键字。这样就将显式声明派生类型的实现故意设计为隐藏父类的版本（同样，在真实世界中，当外部.NET软件和我们当前软件有冲突时，这会很有用）。

```
// 这个类扩展了Circle并隐藏了继承的Draw()方法
class ThreeDCircle : Circle
{
  // 隐藏任何在我之上的Draw()实现
  public new void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

我们还可以把new关键字应用到任何从基类继承的成员类型中（字段、常量、静态方法、属性等）。作为进阶的示例，假设ThreeDCircle希望隐藏继承的PetName属性：

```
class ThreeDCircle : Circle
{
  // 隐藏任何在我之上的PetName属性
  public new string PetName { get; set; }

  // 隐藏任何在我之上的Draw()实现
  public new void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

最后，需要知道，我们仍然可以使用显式强制转换来触发阴影成员在基类中的实现（在下一小节中会介绍）。例如：

```
static void Main(string[] args)
{
...
  // 调用了ThreeDCircle的Draw()方法
  ThreeDCircle o = new ThreeDCircle();
  o.Draw();

  // 调用了父类的Draw()方法
  ((Circle)o).Draw();
  Console.ReadLine();
}
```

源代码 Shapes项目的源代码位于Chapter 6子目录下。

6.6 基类/派生类的转换规则

现在可以构建一组相关的类类型，但需要学习类类型强制转换操作的规则。那么，让我们返回本章之前创建的雇员层次结构。在.NET平台下，系统中的最高基类是System.Object。因此，所有东西都是（“is-a”）object，并且可以照此进行处理。因此，在对象变量中保存任何类型的实例都是合法的：

```
static void CastingExamples()
{
  // Manager是（is-a）System.Object，因此我们刚好可以在对象变量中存储Manager引用
  // 用object变量保存Manager引用也是可以的
  object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
}
```

在Employees系统中，Manager、SalesPerson和PTSalesPerson类型都扩展了Employee，因此我们可以在有效的基类引用中保存任何对象。因此，如下语句也是合法的：

```
static void CastingExamples()
{
  // Manager是（is-a）System.object，因此我们刚好可以在对象变量中存储Manger引用
  object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);

  // Manager同样是Employee
  Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000, "101-11-1321", 1);

  // PTSalesPerson是SalesPerson
  SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000, "111-12-1119", 90);
}
```

类类型之间强制转换的第一条准则就是如果两个类通过“is-a”关系关联，在基类引用中保存派生类型总是安全的。正式地说，这叫做*隐式转换*，因为它基于继承的规则。这就产生了很多强大的编程结构。例如，假设我们在当前Program类中定义了新的方法：

```
static void GivePromotion(Employee emp)
{
  // 增加工资
  // 在公司车库新增停车位

  Console.WriteLine("{0} was promoted!", emp.Name);
}
```

因为这个方法接受单个Employee类型的参数，而且这是“is-a”关系，所以实际上可以将任何Employee类的后代直接传递到这个方法中：

```
static void CastingExamples()
{
  // Manager是（is-a）System.Object，因此我们刚好可以在对象变量中存储Manger引用
  object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);

  // Manager同样是Employee
  Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000, "101-11-1321", 1);
  GivePromotion(moonUnit);

  // PTSalesPerson是SalesPerson
```

6

```
    SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000, "111-12-1119", 90);
    GivePromotion(jill);
  }
```

之前的代码示例从基类类型（Employee）隐式转换为派生类型。然而，如果想解雇Frank Zappa（当前保存在泛型System.Object引用中），该怎么办呢？如果按如下所示把frank对象直接传入GivePromotion()，我们会收到一个编译器错误：

```
// 错误
object frank = new Manager("Frank Zappa", 9, 3000, 40000, "111-11-1111", 5);
GivePromotion(frank);
```

问题在于你传递的不是一个Employee，而是更普遍的System.Object。由于object在继承链中的位置比Employee更高，因此编译器不允许隐式转换，这可以尽可能地保证代码安全。

尽管你能够指出这个object引用指向的是一个内存中与Employee兼容的类，但编译器无法这么做，它直到运行时才能知道这些信息。你可以执行一个显式转换来满足编译器的要求。这就是转换的第二条规则：使用C#强制转换操作符进行显式的向下转换。执行显式转换时所遵循的基本模板如下所示：

```
(ClassIWantToCastTo)referenceIHave
```

因此，要将object变量传递给GivePromotion()方法，必须使用如下的代码：

```
//OK!
GivePromotion((Manager)frank);
```

6.6.1　C#的as关键字

要知道，显式强制转换在运行时而不是编译时进行运算。因此，如果写如下的C#代码：

```
// 我们不能强制转换frank为Hexagon，但编译没问题
Hexagon hex = (Hexagon)frank;
```

可以正确编译，但我们会收到一个运行时错误，或者更正式地说是运行时异常。第7章会研究结构化异常处理的完整细节，然而，现在值得指出的是如果正在进行显式转换，可以通过使用try和catch关键字来捕获可能的无效转换（详细内容参见第7章）：

```
// 捕捉可能的无效转换
try
{
  Hexagon hex = (Hexagon)frank;
}
catch (InvalidCastException ex)
{
  Console.WriteLine(ex.Message);
}
```

虽然这是容错编程的一个好例子，但是C#提供了as关键字在运行时快速检测某个类型是否和另外一个兼容。如果我们使用as关键字，就可以通过检查null返回值来检测兼容性。考虑如下代码：

```
// 使用“as”来测试兼容性
Hexagon hex2 = frank as Hexagon;
if (hex2 == null)
  Console.WriteLine("Sorry, frank is not a Hexagon...");
```

6.6.2 C#的is关键字

由于GivePromotion()方法被设计为接受任何从Employee派生的类型，你可能会想，这个方法怎么检测传到方法内的是哪个派生类型呢？还有，由于传入参数是Employee类型的，我们怎么能访问到SalesPerson和Manager类型的特殊成员呢？

除了as关键字，C#语言还提供了is关键字来检测两个项是否兼容。然而，和as关键字不同的是，如果类型不兼容，is关键字就返回false而不是null引用。考虑如下GivePromotion ()方法的实现：

```
static void GivePromotion(Employee emp)
{
  Console.WriteLine("{0} was promoted!", emp.Name);

  if (emp is SalesPerson)
  {
    Console.WriteLine("{0} made {1} sale(s)!", emp.Name,
      ((SalesPerson)emp).SalesNumber);
    Console.WriteLine();
  }
  if (emp is Manager)
  {
    Console.WriteLine("{0} had {1} stock options...", emp.Name,
      ((Manager)emp).StockOptions);
    Console.WriteLine();
  }
}
```

在这里，我们进行了一个运行时的检查来检测传入的基类引用究竟指向内存中的什么。在检测了是否收到SalesPerson或Manager类型之后，我们就可以进行显式强制转换来获取对类特有成员的访问。还要注意，我们不需要使用try/catch结构来包装我们的强制转换操作，因为我们知道有了这样的条件检测，代码在if区域中的强制转换一定是安全的。

6.7 超级父类：System.Object

在结束本章之前，我想研究一下.NET平台中超级父类的细节：Object。如果读过前几节的话，你可能会注意到，在我们的层次结构（Car、Shape、Employee）中从来就没有显式指定过它们的父类：

```
// 谁是Car的父类
class Car
{...}
```

在.NET世界中，每一个类型最终都会从一个叫System.Object（可以用C#关键字object表示）的基类派生。Object类定义了一组框架中所有类型公共的成员。其实，如果我们构建的类没有显式定义其父类，编译器会自动从Object派生我们的类型。如果我们想很明确地表述自己的意图，完全可以如下定义类来从Object派生：

```
// 从System.Object显式派生
class Car : object
{...}
```

和其他类一样，System.Object定义了一组成员。在如下所示的C#定义中，某些项被声明为virtual，它指定了某个成员可以被子类重写，而其他项标记为static（因此只能在类级别进行调用）：

```
public class Object
{
  // 虚成员
  public virtual bool Equals(object obj);
  protected virtual void Finalize();
  public virtual int GetHashCode();
  public virtual string ToString();

  // 实例级别，非虚成员
  public Type GetType();
  protected object MemberwiseClone();

  // 静态成员
  public static bool Equals(object objA, object objB);
  public static bool ReferenceEquals(object objA, object objB);
}
```

表6-1提供了每一个方法功能的提纲。

表6-1　System.Object的核心成员

对象类的实例方法	作　用
Equals()	默认情况下，如果被比较的项指向内存中同一个项，则方法会返回true。因此，Equals()用于比较对象引用，而不是对象的状态。一般情况下，这个方法会被重写为：如果被比较的对象有相同的内部状态值（也就是基于值的语义），则返回true 要知道，如果重写Equals()，则还需要重写GetHashCode()，因为这些方法在内部用于Hashtable类型从容器获取子对象 回忆在第4章中所介绍的，ValueType类为所有结构重写了该方法，它们进行的比较是基于值的
Finalize()	这个方法（在重写后）在对象销毁之前被调用来释放所有分配的资源。第9章会讨论更多有关CLR的垃圾回收服务
GetHashCode()	这个方法返回int来标识指定的对象实例
ToString()	这个方法使用<namespace>.<type name>格式（叫做完全限定名）返回对象的字符串表示。这个方法可以被子类重写来返回名称/值对的标识字符串以表示对象的内部状态，而不是它的完全限定名
GetType()	这个方法返回Type对象，它完整描述当前指向的对象。简而言之，这是所有对象都可用的运行时类型标识方法（RTTI，在第15章中会讨论更多细节）
MemberwiseClone()	这个方法的作用是逐个成员地返回当前对象的副本，通常用于克隆对象（见第8章）

为了演示Object基类提供的一些默认行为，新建一个叫ObjectOverrides的控制台应用程序。插入一个新的C#类文件，其中包含如下叫Person的空类定义：

```
// 记住！Person扩展Object
class Person {}
```

现在，更新我们的Main()方法来和从System.Object继承的成员进行交互，具体如下所示：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with System.Object *****\n");
```

```
      Person p1 = new Person();

      // 使用System.Object的继承成员
      Console.WriteLine("ToString: {0}", p1.ToString());
      Console.WriteLine("Hash code: {0}", p1.GetHashCode());
      Console.WriteLine("Type: {0}", p1.GetType());

  // 让其他引用指向p1
      Person p2 = p1;
      object o = p2;

      // 引用指向内存中的相同对象吗
      if (o.Equals(p1) && p2.Equals(o))
      {
        Console.WriteLine("Same instance!");
      }
      Console.ReadLine();
    }
  }
```

当前Main()方法的输出结果如下所示：

```
***** Fun with System.Object *****

ToString: ObjectOverrides.Person
Hash code: 46104728
Type: ObjectOverrides.Person
Same instance!
```

首先，注意ToString()的默认实现如何返回当前类型的完全限定名（ObjectOverrides.Person）。在之后对构建自定义命名空间的研究中我们会看到（见第14章），每一个C#项目都定义了“根命名空间”，它和项目本身同名。在这里，我们创建了一个叫ObjectOverrides的项目，因此Person类型（以及Program类）都在ObjectOverrides命名空间内。

Equals()的默认行为用来测试两个变量是不是指向内存中的同一对象。创建一个新的Person变量，名为p1。至此，在托管堆中就有了一个新的Person对象。p2也是Person类型的。然而，我们创建的不是新实例，而是将变量赋值为p1的引用。因此，p1和p2都指向内存中的相同对象，就像变量o（object类型，用来做测试）一样。由于p1、p2和o都指向相同的内存位置，所以相等性测试成功了。

尽管System.Object内置的行为能满足很多需要，但是对我们的自定义类型来说，重写一些继承方法很常见。为了举例说明，更新Person类来支持一些表示个体姓、名字和年龄的状态数据，并且每一个都可以通过自定义构造函数来设置。

```
// 记住，Person扩展了Object
class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public int Age { get; set; }

  public Person(string fName, string lName, int personAge)
  {
    FirstName = fName;
    LastName = lName;
```

```
        Age = personAge;
      }
      public Person(){}
    }
```

6.7.1　重写System.Object.ToString()

我们创建的许多类（和结构）都可以通过重写ToString()返回类型当前状态的字符串表示形式来受益。这对于调试（由于其他原因）来说很有用。至于如何构建这个字符串，就看个人喜好了，然而，推荐的方式是使用分号来分割每一个名称/值对并且在方括号中包装整个字符串（.NET基础类库中的很多类型都遵循这个方式）。下面是Person类重写ToString()的代码：

```
public override string ToString()
{
  string myState;
  myState = string.Format("[First Name: {0}; Last Name: {1}; Age: {2}]",
    FirstName, LastName, Age);
  return myState;
}
```

因为Person类只有3个状态数据，所以ToString()的实现非常直接。然而，始终记住，正确的ToString()重写应该还说明继承链上定义的任何数据。

如果重写ToString()让某个类来扩展自定义基类，第一件事情就是使用base关键字从基类获取ToString()值。获取了父类的字符串数据之后，就可以追加派生类的自定义信息。

6.7.2　重写System.Object.Equals()

让我们也重写Object.Equals()的行为来使用基于值的语义。回忆一下，在默认情况下，只有当被比较的两个对象引用内存中相同的对象实例时才会返回true。对于Person类，两个要比较的变量包含相同状态值时，实现Equals()返回true可能会很有用（例如，姓、名和年龄）。

首先，注意Equals()方法的传入参数是普通System.Object。因此，第一件事就是要确保调用者确实传入了Person对象，并且作为额外的安全措施，还要确保传入参数不是空引用。

只要确定调用者传入的是已分配的Person，实现Equals()的方法就是对传入对象的数据和当前对象的数据进行逐字段的比较。

```
public override bool Equals(object obj)
{
  if (obj is Person && obj != null)
  {
    Person temp;
    temp = (Person)obj;
    if (temp.FirstName == this.FirstName
        && temp.LastName == this.LastName
        && temp.Age == this.Age)
    {
      return true;
    }
    else
    {
      return false;
```

```
      }
    }
    return false;
  }
```

在这里，我们检查传入对象和内部值（使用this关键字）的值。如果两者的名字和年龄相同，我们就认为这两个对象状态相同并因此返回true。任何其他可能性都会返回false。

虽然这个方法确实可行，但是我们想一下，对于一个包含几十个数据字段的类型来说，我们要花多少功夫来实现自定义的Equals()方法啊。一个公共的快捷方式就是利用我们自己的ToString()实现。如果类的ToString()方法正确实现，解释继承链中所有字段数据的话，就只需要比较对象的字符串数据即可：

```
public override bool Equals(object obj)
{
  // 不需要将obj强制转换为Person，因为它们都有ToString()方法
  return obj.ToString() == this.ToString();
}
```

注意，在这种情况下我们没有必要检查传入参数的类型是否正确（本例中为Person），因为.NET中任何类型都支持ToString()方法。更棒的是，我们没有必要逐个检查属性是否相等，因为我们并不是简单地比较ToString()的返回值。

6.7.3 重写System.Object.GetHashCode()

如果一个类重写了Equals()方法，我们还需要重写默认的GetHashCode()实现。简单来说，*散列码*是表示对象某个状态的数字值。例如，如果创建两个字符串对象来保存Hello值，应该得到同样的散列码。然而，如果一个string全是小写（hello），就会得到不同的散列码。

默认情况下，System.Object.GetHashCode()使用对象在内存中的当前位置来产生散列值。然而，如果我们构建要保存在Hashtable类型（在System.Collections命名空间内）中的自定义类型，就应该重写这个成员，因为Hashtable会在内部调用Equals()和GetHashCode()来获取正确的对象。

说明 具体来说，System.Collections.Hashtable类在内部调用GetHashCode()来获取对象的位置，然后（内部）调用Equals()进行精确的匹配。

尽管我们不打算把Person放到System.Collections.Hashtable中，但为了完整性，还是重写一下GetHashCode()。有很多算法可以用来创建散列码，一些很复杂，一些不是那么复杂。大多数情况下，我们可以利用System.String的GetHashCode()实现来生成散列码的值。

由于String类已经有完善的散列码算法来使用String的字符数据计算散列值，因此如果类的某个字段数据与其他所有实例都不相同（如社会保险号码），就只需要对这些字段数据点调用GetHashCode()。这样，如果Person类定义了SSN属性，可以编写如下代码：

```
//假设有一个这样的SSN属性
class Person
{
  public string SSN {get; set;}

  //基于唯一字符串数据点返回散列代码
  public override int GetHashCode()
```

```
    {
      return SSN.GetHashCode();
    }
  }
```

如果不能找到唯一的字符串数据点，而你已经重写了ToString()，可以在自己的字符串表示上调用GetHashcode()：

```
// 根据person的ToString()值返回散列码
public override int GetHashCode()
{
  return this.ToString().GetHashCode();
}
```

6.7.4 测试修改后的Person类

我们已重写了Object的虚成员，现在更新Main()方法来测试我们的更新：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with System.Object *****\n");

  // 说明：我们希望这些都一致，以便测试Equals()和GetHashCode()方法
  Person p1 = new Person("Homer", "Simpson", 50);
  Person p2 = new Person("Homer", "Simpson", 50);

  // 获取对象的串行版本
  Console.WriteLine("p1.ToString() = {0}", p1.ToString());
  Console.WriteLine("p2.ToString() = {0}", p2.ToString());

  // 测试重写的Equals()
  Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));

  // 测试散列码
  Console.WriteLine("Same hash codes?: {0}", p1.GetHashCode() == p2.GetHashCode());
  Console.WriteLine();

  // 修改p2的年龄并再次测试
  p2.Age = 45;
  Console.WriteLine("p1.ToString() = {0}", p1.ToString());
  Console.WriteLine("p2.ToString() = {0}", p2.ToString());
  Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));
  Console.WriteLine("Same hash codes?: {0}", p1.GetHashCode() == p2.GetHashCode());
  Console.ReadLine();
}
```

输出结果如下所示：

```
***** Fun with System.Object *****

p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p1 = p2?: True
Same hash codes?: True

p1.ToString() = [First Name: Homer; Last Name: Simpson; Age: 50]
p2.ToString() = [First Name: Homer; Last Name: Simpson; Age: 45]
p1 = p2?: False
Same hash codes?: False
```

6.7.5 System.Object的静态成员

除了刚才研究的实例级别的成员之外，System.Object还定义了两个（非常有用的）静态成员来测试基于值或基于引用的相等性。考虑如下代码：

```
static void StaticMembersOfObject()
{
  // System.Object的静态成员
  Person p3 = new Person("Sally", "Jones", 4);
  Person p4 = new Person("Sally", "Jones", 4);
  Console.WriteLine("P3 and P4 have same state: {0}", object.Equals(p3, p4));
  Console.WriteLine("P3 and P4 are pointing to same object: {0}",
    object.ReferenceEquals(p3, p4));
}
```

在这里，我们只需要传入两个对象（任何类型）并允许System.Object类自动检测细节。

源代码 ObjectOverrides项目的源代码位于Chapter 6子目录下。

6.8 小结

本章研究了继承和多态的作用和细节。通过这些内容，我们学习了许多支持这些技术的新关键字和标记。例如，冒号用于创建某个类型的父类。父类型可以定义许多虚的或抽象的成员来创建多态接口。派生类型使用override关键字重写这些成员。

除了构建许多类继承之外，本章还研究了如何在基类类型和派生类型之间显式转换，最后我们以.NET基础类库中超级父类System.Object的细节来结束本章的内容。

第7章 结构化异常处理

本章主要讲解使用结构化异常处理（SEH，Structured Exception Handling）来处理C#代码中的运行时异常。读者不仅要学习try、catch、throw、finally等处理异常的C#关键字，还要了解应用程序级异常与系统级异常的区别以及System.Exception基类，其中还将介绍创建自定义异常和如何利用Visual Studio 异常调试工具等主题。

7.1 错误、bug 与异常

尽管有时候有点自负，但我们还是清楚，没有一个程序员是完美的。编写软件是一项复杂的工作，正是由于这种复杂性，即使是最好的软件也经常伴随着各种各样的问题。有时候问题是由糟糕的代码引起的（比如溢出数组的范围），有时候问题又是由应用程序代码库未考虑到的用户错误输入引起的（比如说，将一个电话号码字段赋值为“Chucky”）。不管是何种原因，最终都会导致应用程序无法按照预期正常运行。为了给下面讲述的结构化异常处理做个铺垫，我们首先为3种常用来表述异常的术语下定义。

- **bug**：简单来说，这是由程序员一方引起的错误。举例来说，假定我们在进行非托管C++编程，如果删除已分配的内存失败（从而导致内存泄漏），就会产生一个bug。
- **用户错误**：与bug不同，用户错误往往不是由应用程序作者而是由运行程序的用户引起的。例如，如果你在代码中没有处理错误输入，当最终用户在文本框中输入格式非法的字符串时，很可能会产生用户错误。
- **异常**：异常往往是运行时的非正常情况，在编程时很难被估计到。异常可能包括试图连接一个已经不存在的数据库，打开已被破坏的XML文件，连接当前处于离线状态的机器等。在上述各种情况下，程序员和最终用户都无法完全控制这些异常情况。

通过上面的定义，可以很清楚地看到.NET结构化异常处理是一项很适合处理运行时异常的技术。然而至于我们难以预料的bug和用户错误，CLR常常会引发相应的异常来识别相应的问题。.NET基础类库定义了诸如FormatException、IndexOutOfRangeException、FileNotFoundException、ArgumentOutOfRangeException等数量众多的异常。

在.NET术语命名法中，“异常”解释为bug、用户错误输入和运行时错误，尽管程序员认为这是3个各不相同的问题。在大步前进之前，需要先明确结构化异常处理的作用，同时对比一下它和传统错误处理技术有什么不同之处。

说明 为了使本书中代码示例尽可能简洁，我将不会捕获每种基础类库中指定方法可能引发的异常。在你的产品级项目中，当然需要灵活使用本章中提到的各种技术。

7.2 .NET 异常处理的作用

在.NET之前，Windows操作系统下的错误处理技术是比较混乱的。很多程序员都自己开发错误处理逻辑，只适用于给定的应用程序上下文。例如，一个开发团队可能定义一系列数字常量来表示已知的错误情况，并利用它们作为方法的返回值。我们以下面这段C程序代码为例：

```
/* 一种典型的C语言风格错误捕获机制 */
#define E_FILENOTFOUND 1000

int SomeFunction()
{
  // 假定这个函数中进行的操作导致了如下的返回值
  return E_FILENOTFOUND;
}

void main()
{
  int retVal = SomeFunction();
  if(retVal == E_FILENOTFOUND)
    printf("Cannot find file...");
}
```

由于E_FILENOTFOUND常量比数字常量好不到哪里去，这种方案并不理想，离有效处理这个问题的目标还很远。理想情况下，我们可能希望将这个错误的名称、消息和其他的有用信息都打进一个定义明确的包内（这正是结构化异常处理所做的）。

为处理错误，除了开发人员的临时方案之外，Windows API通过#define、HRESULT等还定义了成百上千的错误代码和基于简单布尔变量的各种变形（bool、BOOL、VARIANT_BOOL等）。此外，许多C++ COM开发人员利用一小组标准COM接口（例如ISupportErrorInfo、IErrorInfo、ICreateErrorInfo）给COM客户端返回有意义的错误信息。

这些早期技术有个明显的问题：极其缺乏对称性。每种方案差不多都是为指定技术、指定语言甚至指定项目而定制的。为了打破这种局面，.NET平台提供了一种标准的技术来发送和捕获运行时错误，这就是结构化异常处理（SEH）。

结构化异常处理方案的优点在于，开发人员现在有了统一的而且对.NET领域内各种语言都通用的方式来处理错误。因此，一个C#程序员处理错误的方法和VB程序员、使用C++/CLI的C++程序员处理错误的方法在语法上相似。

更棒的是，用以引发和捕获异常的语法在不同程序集间或计算机间都是一致的。例如，如果使用C#构建WCF服务，可以向远程调用者抛出一个SOAP错误，而使用的关键字与在同一个应用中的方法之间抛出异常的关键字完全相同。

.NET异常的另一好处是，我们不再是通过接收意义模糊的数字常量来确定问题，而是可以通过异常，它们包含容易读懂的问题描述信息和首次触发异常时调用栈的详细快照。此外，我们还能够为最

终用户提供相应的帮助链接信息，将他们引向一个URL，其中包含相应错误的详细信息和程序员自定义的数据。

7.2.1 .NET异常处理的四要素

结构化异常处理编程要使用4个互相关联的实体：

- 一个表示异常详细信息的类类型；
- 一个向调用者抛出异常类实例的成员；
- 调用者的一段调用异常成员的代码块；
- 调用者的一段处理（或捕获）将要发生异常的代码块。

C#编程语言提供了4个允许我们引发和处理异常的关键字：try、catch、throw和finally。用来表示问题的类型是一个继承自（或派生自）System.Exception的类。下面，了解一下异常处理基类的作用。

7.2.2 System.Exception基类

所有用户定义和系统定义的异常最终都继承自System.Exception基类（它继承自System.Object基类）。这个类的代码如下（请注意其中有些成员是虚的，这样就可能被派生类型重写）：

```
public class Exception : ISerializable, _Exception
{
  // 公有的构造函数
  public Exception(string message, Exception innerException);
  public Exception(string message);
  public Exception();
...

  // 方法
  public virtual Exception GetBaseException();
  public virtual void GetObjectData(SerializationInfo info,
    StreamingContext context);

  // 属性
  public virtual IDictionary Data { get; }
  public virtual string HelpLink { get; set; }
  public Exception InnerException { get; }
  public virtual string Message { get; }
  public virtual string Source { get; set; }
  public virtual string StackTrace { get; }
  public MethodBase TargetSite { get; }
...
}
```

可见，System.Exception定义的很多属性是只读的。很明显，这是由于派生类型将给每个属性提供默认的值。例如，IndexOutOfRangeException类型的默认信息就是“索引超出数组范围”。

说明 Exception类实现了两个.NET接口。尽管我们还没有研究接口（见第8章），只是理解_Exception接口允许非托管代码库（如COM应用程序）处理.NET异常，而ISerializable接口允许异常对象跨边界（如机器边界）持久化。

表7-1详细描述了System.Exception最重要的成员。

表7-1 System.Exception类型的核心成员

System.Exception属性	作 用
Data	此属性返回一个键/值对集合（表示为一个实现IDictionary接口的对象），提供有关该异常的更多程序员定义信息。该集合默认情况下为空
HelpLink	此属性返回一个URL，指向包含详细错误信息描述的帮助文件或网站
InnerException	此属性为只读，可用来获取导致当前异常发生的上一个（或上一组）异常的相关信息。上一个（或上一组）异常作为参数被传入当前异常的构造函数而被记录下来
Message	此属性为只读，它返回指定错误的文字描述。错误信息本身就是构造函数的一个参数
Source	此属性返回引发该异常的程序集或对象的名称
StackTrace	此属性为只读，它包含一个标识触发异常调用序列的字符串。可以想象，在调试过程中，或者要将错误转储到外部错误日志中时，这个属性非常有用
TargetSite	此属性为只读，它返回一个MethodBase对象，其中描述了引发异常的方法的许多细节（ToString()方法将通过名称标识该方法）

7

7.3 最简单的例子

为了阐述结构化异常处理的用处，需要创建一个在适当情况下可能引发异常的类。假定已经创建了一个名为SimpleException的新的控制台应用程序项目，其中定义了两个类——Car和Radio，这两个类之间是拥有（has-a）关系。Radio类定义了一个打开或关闭Radio的方法。

```
class Radio
{
  public void TurnOn(bool on)
  {
    if(on)
      Console.WriteLine("Jamming...");
    else
      Console.WriteLine("Quiet time...");
  }
}
```

除了通过包含/委托来使用Radio类之外，Car类是如此定义的：如果用户加速一个Car对象超过预先定义的最大速度（通过一个名为MaxSpeed的常量成员来指定），Car的引擎会爆炸，使Car不能再用（通过一个名为carIsDead的私有布尔变量成员指定）。

除此之外，Car类型还拥有几个成员变量，分别表示当前速度、用户提供的昵称以及不同的构造函数，用来设置新的Car对象的状态。下面是完整的定义，其中包括代码注释：

```
class Car
{
  // 表示最大速度的常量
  public const int MaxSpeed = 100;

  // Car属性
  public int CurrentSpeed {get; set;}
  public string PetName {get; set;}
```

```
    // Car是否仍可以操纵
    private bool carIsDead;

    // 每个Car拥有一个Radio
    private Radio theMusicBox = new Radio();

    // 构造函数
    public Car() {}
    public Car(string name, int speed)
    {
      CurrentSpeed = speed;
      PetName = name;
    }

    public void CrankTunes(bool state)
    {
      // 委托请求到内部对象
      theMusicBox.TurnOn(state);
    }

    // 查看Car是否过热
    public void Accelerate(int delta)
    {
      if (carIsDead)
        Console.WriteLine("{0} is out of order...", PetName);
      else
      {
        CurrentSpeed += delta;
        if (CurrentSpeed > MaxSpeed)
        {
          Console.WriteLine("{0} has overheated!", PetName);
          CurrentSpeed = 0;
          carIsDead = true;
        }
        else
          Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
      }
    }
  }
```

现在，如果要实现一个如下所示的强制Car对象超过预定义最高速度（在Car类中，设为100）的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Exception Example *****");
  Console.WriteLine("=> Creating a car and stepping on it!");
  Car myCar = new Car("Zippy", 20);
  myCar.CrankTunes(true);

  for (int i = 0; i < 10; i++)
    myCar.Accelerate(10);
  Console.ReadLine();
}
```

将会看到如下所示的输出结果：

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90
=> CurrentSpeed = 100
Zippy has overheated!
Zippy is out of order...
```

7.3.1 引发普通的异常

现在有了可以工作的Car类，下面阐述引发一个异常的最简单方法。当前实现的Accelerate()方法在调用者试图加速到Car对象的速度上限以上时，显示一条错误信息。

对此方法做些改动，如果用户在车坏了后还试图加速汽车，就会引发异常，我们将创建并设定一个新的System.Exception类的实例，并通过类的构造函数为只读属性Message赋值。如果将异常对象返回被调用者，可以使用C#中的throw关键字。下面是新Accelerate()方法的相关代码：

```
// 这次如果用户加速到超过最大速度，就会引发异常
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;

      // 使用throw关键字引发异常
      throw new Exception(string.Format("{0} has overheated!", PetName));
    }
    else
      Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

在检查调用者如何捕获这个异常之前，有很多有意思的地方需要注意。首先，如果我们引发一个异常，总是由我们来决定所引发的问题和何时引发异常。这里，假定程序试图加速一辆已失效的坏车，就要引发一个System.Exception对象，以表示Accelerate()方法无法继续（事实上这个假定可能是无效的，需要根据你创建的应用程序来决定）。

另一种可供选择的方案是，可以在Accelerate()方法中实现自动恢复，而无需立即引发异常。一般情况下，异常应当仅仅在一个较为致命的条件满足后引发（比如未发现必要的文件，连接数据库失败等）。决定什么条件下引发异常是我们必须应对的一个设计问题。当前假定要求对这个已坏的汽车继续加速，以构成一个引发异常的条件。

7.3.2 捕获异常

因为Accelerate()方法现在引发了异常，调用者需要做好准备来处理可能发生的异常。当调用一个可能引发异常的方法时，应当使用try/catch块。一旦捕获到异常对象，将能够调用异常对象成员来释放问题的详细信息。

如何处理这份数据取决于我们自己。你可能希望将其记录到报告文件里，写进Windows事件日志里，发电子邮件给系统管理员或者将问题显示给最终用户。这里，我们就将其输出到控制台窗口上：

```
// 处理引发的异常
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Exception Example *****");
  Console.WriteLine("=> Creating a car and stepping on it!");
  Car myCar = new Car("Zippy", 20);
  myCar.CrankTunes(true);

  // 加速车到超过最大速度以触发异常
  try
  {
    for(int i = 0; i < 10; i++)
      myCar. Accelerate(10);
  }
  catch(Exception e)
  {
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Method: {0}", e.TargetSite);
    Console.WriteLine("Message: {0}", e.Message);
    Console.WriteLine("Source: {0}", e.Source);
  }

  // 异常被处理了，转到下一个语句
  Console.WriteLine("\n***** Out of exception logic *****");
  Console.ReadLine();
}
```

其实，try块是执行过程中可能引发异常的语句的一部分。如果检测到一个异常，程序执行流进入相应的catch块中。另一方面，如果try块内包含的代码没有触发异常，相应的catch代码块就被直接略过，说明一切正常。下面的输出结果显示了运行这个程序的一个测试。

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Method: Void Accelerate(Int32)
Message: Zippy has overheated!
Source: SimpleException

***** Out of exception logic *****
```

正如你所见，当一个异常经过处理后，应用程序将继续执行catch块之后的代码。在有些情况下，一个异常可能严重到可以终止程序的运行。然而大多数情况下，异常处理包含的逻辑将保证应用程序顺畅地运行（尽管它可能丧失了部分功能，比如无法链接到远程数据源）。

7.4 配置异常的状态

现在，Accelerate()方法内配置的System.Exception对象已经通过构造函数参数创建了一个公开给Message属性的值。然而，如前面的表7-1所示，Exception类还会提供其他成员（TargetSite、StackTrace、HelpLink和Data），这些成员在需要进一步界定问题本质时很有用。为了完善当前的示例，让我们逐一深入查看这些成员的细节。

7.4.1 TargetSite属性

System.Exception.TargetSite属性帮助我们了解引发某个异常的方法的各种信息。就像在前面Main()方法中呈现的那样，输出TargetSite的值将显示返回值类型、名称、引发异常方法的参数。可是，TargetSite不是只返回华而不实的字符串，而是返回一个强类型的System.Reflection.MethodBase对象。这种类型可用于收集引发异常的方法以及定义引发异常的方法的类的许多信息。假定我们的上一个catch逻辑更新为如下代码：

```
static void Main(string[] args)
{
...
  // TargetSite实际上返回一个MethodBase对象
  catch(Exception e)
  {
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Member name: {0}", e.TargetSite);
    Console.WriteLine("Class defining member: {0}",
      e.TargetSite.DeclaringType);
    Console.WriteLine("Member type: {0}", e.TargetSite.MemberType);
    Console.WriteLine("Message: {0}", e.Message);
    Console.WriteLine("Source: {0}", e.Source);
  }
  Console.WriteLine("\n***** Out of exception logic *****");
  Console.ReadLine();
}
```

在这里，我们使用MethodBase.DeclaringType属性值来指定引发异常的类的全称（这个例子中为SimpleException.Car），使用MethodBase对象的MemberType属性来确定引发异常的成员类型（比如属性与方法）。这里catch逻辑块的输出结果如下：

```
*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
```

7

7.4.2 StackTrace属性

System.Exception.StackTrace属性帮助我们标识引发异常的一系列调用。需要注意的是，StackTrace的值是异常创建时自动产生的，无法为其赋值。假定我们再次更改catch逻辑，如下所示：

```
catch(Exception e)
{
  ...
  Console.WriteLine("Stack: {0}", e.StackTrace);
}
```

如果运行这个程序，你将发现下面的栈跟踪被输出到控制台（当然你会发现这里的行号和应用程序文件夹路径可能与你的有所不同）：

```
Stack: at SimpleException.Car.Accelerate(Int32 delta)
in c:\MyApps\SimpleException\car.cs:line 65 at SimpleException.Program.Main()
in c:\MyApps\SimpleException\Program.cs:line 21
```

StackTrace返回的字符串证明，正是这个调用序列引发了异常。请注意下面行号字符串标识这个序列的首次调用，上面行号字符串标识出错成员的具体位置。很明显，在调试或记录指定应用程序时，这些信息都非常有用，它使我们能顺其自然地发现错误的根源。

7.4.3 HelpLink属性

尽管TargetSite属性和StackTrace属性能够帮助程序员了解指定的异常，它们对最终用户而言却基本上没什么用。你已经看到，System.Exception.Message属性可用来获取呈现给当前用户的可以阅读的信息。除此之外，HelpLink属性能帮助用户找到具体的URL或包含更详细相关信息的标准Windows帮助文件。

默认情况下，HelpLink属性的值是一个空字符串。如果读者需要用一个有意义的值填充该属性，就要在引发System.Exception类型异常之前赋值。下面是对Car.Accelerate()方法所做的相应更改：

```
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;

      // 我们需要调用HelpLink属性，因此需要在异常对象引发之前先创建一个本地变量
      Exception ex =
        new Exception(string.Format("{0} has overheated!", PetName));
      ex.HelpLink = "http://www.CarsRUs.com";
```

```
        throw ex;
      }
      else
        Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
    }
  }
```

catch逻辑块将被更新，以输出如下的帮助链接信息：

```
catch(Exception e)
{
  ...
  Console.WriteLine("Help Link: {0}", e.HelpLink);
}
```

7.4.4 Data属性

System.Exception中的Data属性允许我们使用用户提供的相应信息（如时间戳）来填充异常对象。Data属性返回一个实现了定义在System.Collections命名空间下的IDictionary接口的对象。第8章将讨论基于接口的编程和System.Collections命名空间的作用，现在暂时了解dictionary集合允许创建一组使用指定键检索的值即可。请观察再次更新过的Car.Accelerate()方法：


```
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", PetName);
  else
  {
    CurrentSpeed += delta;
    if (CurrentSpeed >= MaxSpeed)
    {
      carIsDead = true;
      CurrentSpeed = 0;

      // 我们需要调用HelpLink属性，因此需要在异常对象引发之前先创建一个本地变量
      Exception ex =
        new Exception(string.Format("{0} has overheated!", PetName));
      ex.HelpLink = "http://www.CarsRUs.com";

      // 填充关于错误的自定义数据
      ex.Data.Add("TimeStamp",
        string.Format("The car exploded at {0}", DateTime.Now));
      ex.Data.Add("Cause", "You have a lead foot.");
      throw ex;
    }
    else

    Console.WriteLine("=> CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

为了成功列举键/值对，需要在包含实现Main()方法的类的文件中使用DictionaryEntry类型，不过首先必须确保为System.Collections命名空间指定了using指令：

```
using System.Collections;
```

下一步，需要更新catch块的逻辑来测试Data属性的返回值不为null（默认值）。在此之后，利用

DictionaryEntry类型的Key和Value属性输出自定义数据到控制台：

```
catch (Exception e)
{
...
  // 默认情况下，data字段是空的，需要检查它是否为空
  Console.WriteLine("\n-> Custom Data:");
  if (e.Data != null)
  {
    foreach (DictionaryEntry de in e.Data)
      Console.WriteLine("-> {0}: {1}", de.Key, de.Value);
  }
}
```

完成这些之后，可以看到如下所示的输出结果：

```
***** Simple Exception Example *****
=> Creating a car and stepping on it!
Jamming...
=> CurrentSpeed = 30
=> CurrentSpeed = 40
=> CurrentSpeed = 50
=> CurrentSpeed = 60
=> CurrentSpeed = 70
=> CurrentSpeed = 80
=> CurrentSpeed = 90

*** Error! ***
Member name: Void Accelerate(Int32)
Class defining member: SimpleException.Car
Member type: Method
Message: Zippy has overheated!
Source: SimpleException
Stack: at SimpleException.Car.Accelerate(Int32 delta)
       at SimpleException.Program.Main(String[] args)
Help Link: http://www.CarsRUs.com

-> Custom Data:
-> TimeStamp: The car exploded at 1/12/2010 8:02:12 PM
-> Cause: You have a lead foot.

***** Out of exception logic *****
```

Data属性非常有用，因为它允许我们打包关于“错误”的自定义信息，无需构建全新的类类型来扩展Exception基类。然而和Data属性一样有用的是构建强类型的异常类，因为.NET开发人员可以通过强类型属性来使用自定义数据。

这个方法允许调用者捕获Exception派生的类型，而无需深入数据集合来获取其他细节。要理解为什么这么做，我们还需要研究系统级别异常和应用级别异常的区别。

源代码 SimpleException项目的源代码位于Chapter 7子目录下。

7.5 系统级异常

.NET基础类库定义了许多派生自System.Exception的类。例如，在System命名空间下定义的核心错误对象有ArgumentOutOfRangeException、IndexOutOfRangeException、StackOverflowException等。其他命名空间下定义了反映该命名空间行为的异常，比如说System.Drawing.Printing定义了输出异常，System.IO定义了基于输入输出的异常，System.Data定义了数据库异常及数据库相关异常，等等。

准确地说，.NET平台引发的异常应被称为*系统异常*。这些异常被认为是无法修复的致命错误。系统异常直接派生自名为System.SystemException的基类，该基类派生自System.Exception，而后者又派生自System.Object：

```
public class SystemException : Exception
{
  // 各种构造函数
}
```

既然System.SystemException类型除了一组自定义构造函数外不添加任何功能，读者首先想到的可能是，系统异常的存在有什么必要性呢？很简单，当一个异常类型派生自System.SystemException时，我们就能够判断引发该异常的实体是.NET运行库而不是正在执行的应用程序代码库。可以通过is关键字来验证这个结论：

```
// NullReferenceException是系统异常
NullReferenceException nullRefEx = new NullReferenceException();
Console.WriteLine("NullReferenceException is-a SystemException? : {0}",
                  nullRefEx is SystemException);
```

7.6 应用程序级异常

既然所有的.NET异常都是类类型，我们就可以随意创建应用程序特定的异常了。由于System.SystemException基类表示从CLR引发的所有异常，读者可能很自然地准备将所有自定义异常派生自System.Exception类型。事实上最佳实践表明，自定义异常应当派生自System.ApplicationException类型：

```
public class ApplicationException : Exception
{
  // 各种构造函数
}
```

就像系统异常一样，应用程序异常并不在一组构造函数外再定义其他任何成员。从功能上来讲，System.ApplicationException的唯一目的就是标识出错误的来源。当读者处理一个派生自System.ApplicationException的异常时，可以设想异常是由正在执行的应用程序代码库引发的，而不是由.NET基础类库或.NET运行时引擎引发的。

说明 事实上，很少有.NET开发人员会创建扩展ApplicationException的自定义异常。而更常见的情况则是将其简单地归入System.Exception类，不过这两种方法在技术上都是合法的。

7.6.1　构建自定义异常，第一部分

正如第一个例子中所示，读者可以一直引发System.Exception的实例来标识运行时错误，但有时候构建一个强类型异常来表示当前问题的独特细节更好。举个例子，假定要构建一个名为CarIsDeadException的自定义异常来表示加速注定要损坏的汽车的错误。第一步就是创建一个派生自System.Exception/System.ApplicationException的新类（按照约定，所有的异常类均应以“Exception”后缀结束，这是.NET的最佳实践）。

说明　作为一个规则，所有自定义异常类都应该定义为公共类（回忆一下，非嵌套类型的默认访问修饰符是内部的）。这是因为异常通常都会跨程序集边界进行传递，也应该可以被调用代码库所访问。

新建一个控制台应用程序CustomException，使用Project→Add Existing Item菜单项将之前的Car.cs和Radio.cs复制到新项目中（请确保将定义Car和Radio类型的命名空间由SimpleException改为CustomException）。然后，增加如下的类定义：

```
// 这个自定义异常描述了car-is-dead条件下的详细信息（记住，也可以只是扩展异常）
public class CarIsDeadException : ApplicationException
{}
```

和其他类一样，我们可以自由包含能在调用逻辑catch块中调用的任意数量的自定义成员，也可以自由重写任何父类定义的虚拟成员。例如，可以通过重写虚拟Message属性来实现CarIsDeadException。

同样，构造函数没有在抛出异常时填充数据字典（通过Data属性），而是允许调用者传入时间戳和错误原因。最后，我们可以通过使用强类型属性来获取时间戳和错误原因：

```
public class CarIsDeadException : ApplicationException
{
  private string messageDetails = String.Empty;
  public DateTime ErrorTimeStamp {get; set;}
  public string CauseOfError {get; set;}

  public CarIsDeadException(){}
  public CarIsDeadException(string message,
    string cause, DateTime time)
  {
    messageDetails = message;
    CauseOfError = cause;
    ErrorTimeStamp = time;
  }

  // 重写Exception.Message属性
  public override string Message
  {
    get
    {
      return string.Format("Car Error Message: {0}", messageDetails);
    }
  }
}
```

这里，CarIsDeadException类包含一个表示当前异常的私有数据成员（messageDetails），可以通过自定义构造函数来设定其值。从Accelerate()引发异常很直接，只需分配、配置和引发一个CarIsDeadException类型，而不是通用的System.Exception异常（在本例中，不需要手工填充数据集合）：

```
// 抛出自定义CarIsDeadException
public void Accelerate(int delta)
{
...
  CarIsDeadException  ex =
   new CarIsDeadException (string.Format("{0} has overheated!", PetName),
                           "You have a lead foot", DateTime.Now);
  ex.HelpLink = "http://www.CarsRUs.com";
  throw ex;
...
}
```

为了捕获到传入的异常，我们调整catch的范围为特定的CarIsDeadException类型（由于System.Exception包含CarIsDeadException，所以仍然可以在System.Exception中捕获到该异常）：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);

  try
  {
    // 行程异常
    myCar.Accelerate(50);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
    Console.WriteLine(e.ErrorTimeStamp);
    Console.WriteLine(e.CauseOfError);
  }
  Console.ReadLine();
}
```

既然读者理解了构建自定义异常的基本过程，可能想知道什么时候需要构建自定义异常。通常情况下，读者仅需在出现错误的类与该错误关系紧密时才需要创建自定义异常（例如，一个自定义文件类引发许多文件相关的错误，一个Car类引发许多汽车相关的错误，一个数据访问对象引发关于特定数据库表的错误，等等）。这样我们就能使调用者逐个地处理众多的异常。

7.6.2 构建自定义异常，第二部分

为了配置自定义错误信息，当前的CarIsDeadException类型重写了System.Exception.Message属性，并提供两个自定义属性来说明其他数据。事实上，不用重写Message虚属性，而只需要将传入的信息按以下方式传递给父对象的构造函数：

```
public class CarIsDeadException : ApplicationException
{
  public DateTime ErrorTimeStamp { get; set; }
  public string CauseOfError { get; set; }
```

```
    public CarIsDeadException() { }

    // 将信息传递给父对象构造函数
    public CarIsDeadException(string message, string cause, DateTime time)
      :base(message)
    {
      CauseOfError = cause;
      ErrorTimeStamp = time;
    }
  }
```

请注意，这次我们并没有定义一个字符串变量来呈现信息，也没有重写Message属性，仅是将参数传递到基类构造函数而已。通过这样的设计，一个自定义异常类就没有任何基类重写，和一个派生自System.ApplicationException的特定命名的类没有任何差别了。

要知道大多数（甚至全部）的自定义异常类都遵循这个简单的模式，请不要感到惊讶。很多情况下，自定义异常类的作用并不是提供继承基类之外附加的功能，而是提供明确标识错误种类的强命名类型，因此客户会为不同类型的异常提供不同的处理程序逻辑。

7.6.3　构建自定义异常，第三部分

如果读者想构造一个真正意义上严谨规范的自定义异常类，需要确保类遵守.NET异常处理的最佳实践。具体来讲，自定义异常需要：

- 继承自Exception/ApplicationException类；
- 有[System.Serializable]特性标记；
- 定义一个默认的构造函数；
- 定义一个设定继承的Message属性的构造函数；
- 定义一个处理“内部异常”的构造函数；
- 定义一个处理类型序列化的构造函数。

基于读者当前对.NET的了解，可能完全不知道特性标记或对象序列化的作用，不过没关系，我将在本书稍后介绍这些主题（第15章讨论特性，第20章讨论序列化服务）。为了完成我们自定义异常的构建，下面是CarIsDeadException最终的完整代码，它说明了每个特殊的构造函数（其他自定义属性和构造函数详见7.6.2节）：

```
[Serializable]
public class CarIsDeadException : ApplicationException
{
  public CarIsDeadException() { }
  public CarIsDeadException(string message) : base( message ) { }
  public CarIsDeadException(string message,
                            System.Exception inner)
    : base( message, inner ) { }
  protected CarIsDeadException(
    System.Runtime.Serialization.SerializationInfo info,
    System.Runtime.Serialization.StreamingContext context)
    : base( info, context ) { }

  // 其他自定义属性、构造函数、数据成员……
}
```

既然遵循.NET最佳实践构建的自定义异常只在名称上有区别，你一定乐意知道Visual Studio提供了一个叫做“Exception”的代码片段模板（如图7-1所示），它能自动生成一个新的遵循.NET最佳实践的异常类［在第2章中我们介绍过，可以通过输入名称（本例中为exception）并连按两次Tab键来激活代码块］。

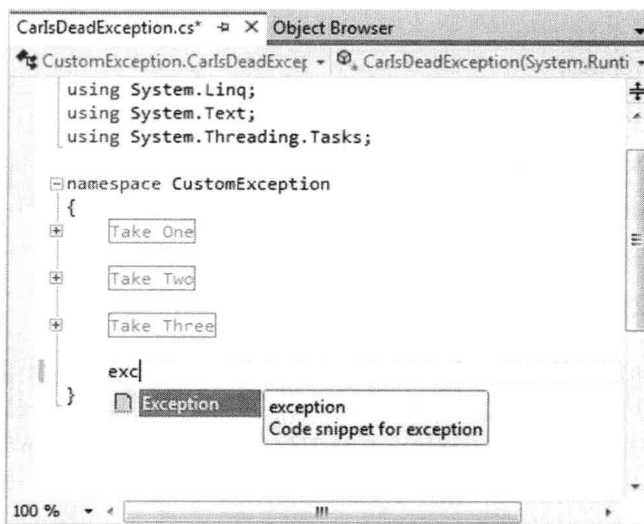


图7-1 Exception代码片段模板

源代码 CustomException项目的源代码位于Chapter 7子目录下。

7.7 处理多个异常

在最简单的情形下，一个try块有一个catch块。但在现实中，你常常会遇到包含try块的语句能够触发许多可能发生的异常的情形。创建一个名为ProcessMultipleExceptions的新控制台应用程序项目，通过Project→Add Existing Item将已有的Car.cs、Radio.cs和CarIsDeadException.cs文件添加到新项目中，同时相应地修改命名空间的名字。

例如，假定在读者传递了一个无效参数（比如小于0的任何值）的情况下，修改Car的Accelerate()方法还会引发一个基础类库预定义的异常ArgumentOutOfRangeException。注意，该异常类的构造函数所接收的第一个字符串为错误参数的名称，然后是描述该错误的消息。

```
// 在处理之前传入无效参数的测试
public void Accelerate(int delta)
{
  if(delta < 0)
    throw new
      ArgumentOutOfRangeException("delta", "Speed must be greater than zero!");
  ...
}
```

catch逻辑现在可以为每种异常分别做出回应：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // 触发超出范围的异常
    myCar.Accelerate(-10);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

当读者创建多个catch块的时候，必须注意一个异常引发后都将被第一个可用的catch处理。为了阐明什么是第一个可用的catch块，假定我们修改之前的逻辑来增加另外一个catch块，如下所示，它试图处理通过捕获普通的System.Exception来处理包括CarIsDeadException和ArgumentOutOfRange-Exception在内的所有异常：

```
// 这段代码将不会编译
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);

  try
  {
    // 触发超出范围的异常
    myCar.Accelerate(-10);
  }
  catch(Exception e)
  {
    // 处理其他的所有异常
    Console.WriteLine(e.Message);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

这段异常处理逻辑产生了编译时错误。问题出在（由于“is-a”关系）第一个catch块可以处理任何派生自System.Exception的异常，其中包括CarIsDeadException与ArgumentOutOfRangeException，故而最终导致无法到达另外两个catch块！

读者要牢记的首要原则就是，要保证将catch块按照下面的原则结构化：最前面的catch捕获最特定的异常（也就是一个异常类型派生关系链中排在最上面的派生类型），最后面的catch捕获最普通的异常（也就是指定异常继承关系链中的基类，在本例中是System.Exception）。

这样，如果读者想定义一个在CarIsDeadException与ArgumentOutOfRangeException之后能捕获任何错误的catch语句，应当这样写：

```
// 这段代码编译正常
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    // 触发超出范围的异常
    myCar.Accelerate(-10);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  // 捕获CarIsDeadException和ArgumentOutOfRangeException之后的所有异常
  catch (Exception e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

说明 人们通常尽可能地捕获特定的异常类，而不是一般的System.Exception。这表面上看起来很省事（你可能会想“啊，它可以捕获所有我不关心的事情”），但从长期来看，很可能导致莫名其妙的运行时崩溃，因为有一个更严重的错误你没有在代码中处理。记住，往往在最后一个catch块中才处理System.Exception。

7.7.1 通用的catch语句

C#也支持通用catch块，它不显式接收由指定成员引发的异常对象：

```
// 通用的catch
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    myCar.Accelerate(90);
  }
  catch
```

```
    {
      Console.WriteLine("Something bad happened...");
    }
    Console.ReadLine();
  }
```

很明显，这不是处理异常的最佳途径，因为我们无法获取关于发生错误的有意义的资料（比如说方法名、调用栈或自定义信息）。不过，在C#中这样的构造确实是允许的，通过它可以用通用模式来处理所有错误。

7.7.2 再次引发异常

请注意，可以在try块逻辑中向之前的调用者再次引发一个调用栈异常。要想这样，仅仅在catch块中使用throw关键字就行了，它通过调用逻辑链传递异常。在catch块只能处理即将发生的部分错误时这样做很有用：

```
// 传递异常
static void Main(string[] args)
{
...
  try
  {
    // 给汽车加速的逻辑
  }
  catch(CarIsDeadException e)
  {
    // 执行一些处理此错误的操作并传递异常
    throw;
  }
...
}
```

注意，在这段示例代码中，CarIsDeadException传给了CLR，该异常由Main()方法再次引发。这样的话，最终用户将看到一个系统提供的错误对话框。通常情况下，只需要向调用者再次引发部分处理过的异常，使之能够恰当地处理即将发生的异常。

还要注意，我们没有显式重新抛出CarIsDeadException对象，而是使用了不带参数的throw关键字。我们并没有创建新的异常对象，只是重新抛出了原始的异常对象（及其原始信息）。这样做保护了原始目标的上下文。

7.7.3 内部异常

我们完全可以在处理其他异常的时候触发一个异常。例如，假定我们在一个特定的catch块中处理CarIsDeadException，在这个过程中试图将栈跟踪记录到C盘下名为carError.txt的文件中（必须明确指定通过System.IO命名空间来使用这些I/O类型）：

```
catch(CarIsDeadException e)
{
  // 试图打开C盘下名为carErrors.txt的文件
  FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
  ...
}
```

如果指定的文件并不位于C盘中，调用File.Open()将导致一个FileNotFoundException！本书稍后会介绍关于System.IO命名空间的一切,并介绍在首次试图打开文件之前如何通过编程判断硬盘上该文件存在与否（总而言之这个异常可以避免）。为了将注意力集中到关于异常的主题上来，我们假定这个异常已经产生。

如果在处理一个异常的时候遇到另一个异常，最好的习惯是将这个新异常对象标识为与第一个异常类型相同的新对象中的“内部错误”，这个建议比较拗口。我们之所以需要创建一个异常的新对象来等待处理，是因为声明一个内部错误的唯一途径就是将其作为一个构造函数参数。考虑以下代码：

```
catch (CarIsDeadException e)
{
  try
  {
    FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
    ...
  }
  catch (Exception e2)
  {
    // 引发记录新异常的异常，还有第一个异常的相关信息
    throw new CarIsDeadException(e.Message, e2);
  }
}
```

注意在本例中，我们将FileNotFoundException对象作为CarIsDeadException的第二个参数传递进来。一旦确定了这个新对象，我们就向下一个调用者的调用栈引发异常，本例中下一个调用者是Main()方法。

既然Main()方法没有下一个调用者来捕获异常，我们需要通过错误对话框将其再次呈现出来。就像再次引发异常一样，记录内部异常通常仅仅在调用者能够在首次发生异常时将其更恰当地捕获处理的情况下才有用。例如，在本例中如果调用者的catch块逻辑能够利用InnerException属性来获取内部异常对象的详细信息的话，记录内部异常才有用。

7.7.4　finally块

一个try/catch块后面可能接着会定义一个finally块。finally块并不是必须要有的，它是为了保证不管是否有异常（或其他任何类型），一组代码语句始终都能被执行。假定在退出Main()方法前，不论任何异常处理发生与否，读者总是要先关掉车上面广播的电源：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  myCar.CrankTunes(true);

  try
  {
    // 车的加速逻辑
  }
  catch(CarIsDeadException e)
  {
    // 处理CarIsDeadException
  }
  catch(ArgumentOutOfRangeException e)
  {
```

7

```
      // 处理ArgumentOutOfRangeException
    }
    catch(Exception e)
    {
      // 处理其他任何异常
    }
    finally
    {
      // 不论异常发生与否，以下语句总是被执行
      myCar.CrankTunes(false);
    }
    Console.ReadLine();
  }
```

如果读者不加入一个finally块，当异常发生时（很可能）广播将不会被关闭。在更加现实的场景中，当读者准备进行销毁对象、关闭文件、断开数据库连接等操作时，将资源清理加入到finally块来确保操作正确执行。

7.8　谁在引发什么异常

既然.NET Framework中的方法在各种情况下可以引发任意数量的异常，很自然会产生一个疑问："我怎么知道指定的基础类库中的方法可能引发哪个异常呢？"最终的答案就是：参阅.NET Framework 4.5 SDK文档。帮助系统文档中的每个方法都有指定成员可能引发的异常。另外一个可供选择的办法是：在Visual Studio 中可以通过悬停鼠标光标在代码窗口的成员名称上，来浏览该基础类库成员可能引发的所有异常（如果有的话）的列表，如图7-2所示。

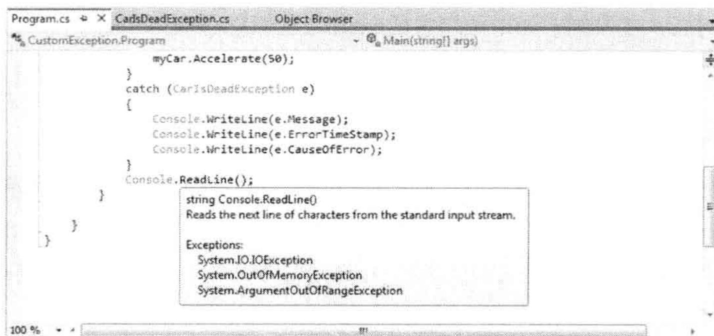


图7-2　确定一个指定方法中可能引发的所有异常

说明　对于由Java转向.NET的程序员来说，要理解类型成员不是一系列可能引发异常的原型（换句话说，.NET不支持检查异常）。不管怎样，读者不需要逐个处理指定成员可能引发的所有异常。

7.9　未处理异常的后果

在这里，读者可能会思考，如果一个直接被引发的异常未被处理，会发生什么。假定Main()逻辑

中我们将Car对象增速到超过速度上限而不使用try/catch逻辑：

```
static void Main(string[] args)
{
    Console.WriteLine("***** Handling Multiple Exceptions *****\n");
    Car myCar = new Car("Rusty", 90);
    myCar.Accelerate(500);
    Console.ReadLine();
}
```

忽视这个异常将呈现一个“未处理的异常”对话框，非常妨碍最终用户使用我们的程序，如图7-3所示。

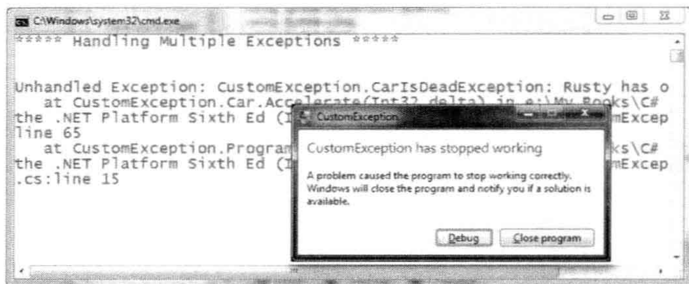


图7-3 不处理异常的后果

7.10 使用 Visual Studio 调试未处理的异常

请注意，为了解决这些问题，Visual Studio提供了大量工具来帮助我们调试未处理自定义异常。再次假定将Car对象加速超过了速度上限。如果启动了调试会话（选择Debug→Start Debugging菜单），Visual Studio会在未处理异常引发时自动中断。现在好了，一个显示Message属性值的窗口呈现在我们面前，如图7-4所示。

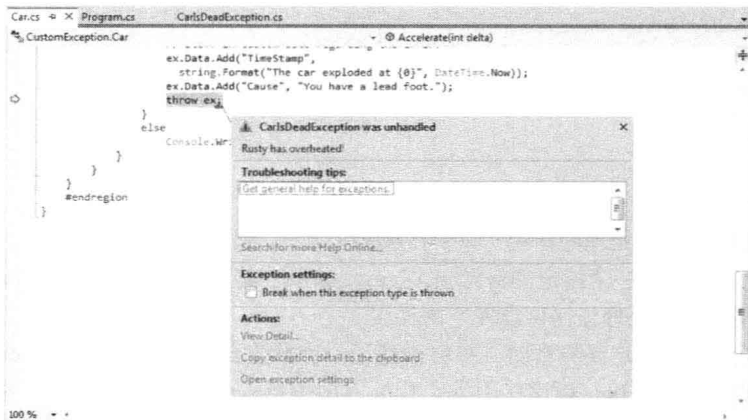


图7-4 使用Visual Studio调试未处理的自定义异常

说明　如果没有成功处理由.NET基础类库中方法引发的异常，Visual Studio调试工具将在调用该出错方法的语句处中断。

单击View Detail链接，能够看到该对象状态的详细信息，如图7-5所示。

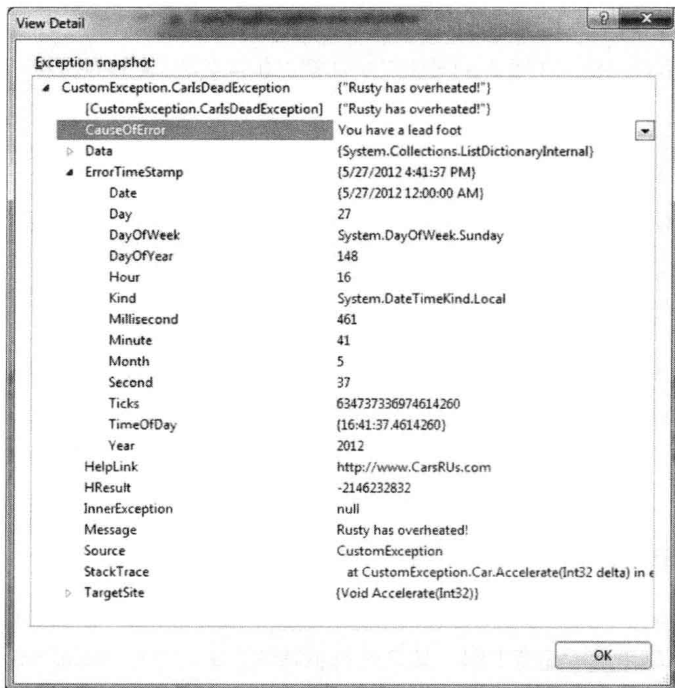


图7-5　查看异常详细信息

源代码　ProcessMultipleExceptions项目的源代码位于Chapter 7子目录下。

7.11　小结

本章介绍了结构化异常处理的作用。当一个方法需要发送错误对象给调用者时，通过C#中的`throw`关键字来分配、设置并引发一个特定的`System.Exception`派生类型。通过使用C#中`catch`关键字和可选的`finally`块，调用者能够处理任何可能发生的错误。

当读者自己创建一个自定义异常时，最终创建了一个派生自`System.ApplicationException`的类，由它来表示一个异常从当前执行程序中引发。相反，派生自`System.SystemException`的错误对象呈现CLR引发的重要（和致命的）错误。最后，本章介绍了各种可用于依据.NET最佳实践创建自定义异常和调试异常的Visual Studio工具。

第8章

接　口

8

本章讨论基于接口的编程，进一步加深我们对面向对象开发的理解。我们将会学习如何定义和实现接口，进而理解构建支持多种行为的类型有哪些优势。而后，会讨论一系列相关主题，如获取接口引用、显式接口实现以及接口层次结构的构建。我们还会学习许多定义在.NET基础类库中的标准接口。我们会看到，自定义类和结构完全可以实现这些预定义的接口，以支持对象克隆、对象枚举和对象排序等高级行为。

8.1　接口类型

首先给出接口类型的正式定义。接口就是一组抽象成员的命名集合。回忆一下第6章，抽象方法是纯粹的协议，在其中没有提供默认的实现。由接口定义的某个特定成员依赖于它所模拟的确切行为。是的，接口表示某个类或结构可以选择去实现的行为。此外，在本章中我们会看到，一个类（或者一个结构）可以支持任意数量的接口，因此本质上也就支持了多种行为。

.NET基础类库提供了几百个预定义的接口类型，由各种类和结构实现。例如，在第21章中我们会看到，ADO.NET附带了多个数据提供程序，允许我们和某个数据库管理系统进行交互。因此，在ADO.NET下，我们有很多连接对象可以选择（SqlConnection、OleDBConnection、OdbcConnection等）。

尽管每一个连接对象都有唯一的名字，也定义在不同的命名空间中，某些可能还在不同的程序集中，但是所有连接对象都实现了一个叫IDbConnection的公共接口：

```
// IDbConnection接口定义了一组所有连接对象都支持的公共成员
public interface IDbConnection : IDisposable
{
  // 方法
  IDbTransaction BeginTransaction();
  IDbTransaction BeginTransaction(IsolationLevel il);
  void ChangeDatabase(string databaseName);
  void Close();
  IDbCommand CreateCommand();
  void Open();

  // 属性
  string ConnectionString { get; set;}
  int ConnectionTimeout { get; }
  string Database { get; }
  ConnectionState State { get; }
}
```

说明　根据惯例，.NET接口多以大写字母“I”作为前缀。如果我们创建自定义接口，可依照此最佳实践如法炮制。[①]

现在不要太过关注这些成员是做什么的。只需要理解IDbConnection接口定义了一组所有ADO.NET连接对象都共有的成员。因此，也就保证了每一个连接对象都支持诸如Open()、Close()、CreateCommand()等的成员。此外，由于接口成员总是抽象的，每一个连接对象完全可以以自己的方式来实现这些方法。

继续看本书的话，你会找到许多.NET基础类库提供的接口。我们自定义的类和结构中可以实现这些接口，以定义各种类型，与整个框架紧密融合起来。

对比接口类型和抽象基类

如果你看过第6章的话，就会发现接口类型和抽象基类很相似。回忆一下，如果类被标记为抽象的，它可以定义许多抽象成员来为所有派生类型提供多态接口。然而，虽然类定义了一组抽象成员，它完全可以再定义许多构造函数、字段数据、非抽象成员（具有实现）等。而接口，只能包含抽象成员。

由抽象父类创建的多态接口有一个主要的限制，那就是只有派生类型才支持由抽象父类定义的成员。然而，在大型软件系统中，开发除了System.Object之外没有公共父类的多个类层次结构很普遍。由于抽象基类中的抽象成员只应用到派生类型，我们就不能以多个层次结构配置类型来支持相同的多态接口。作为示例，假设我们已经定义了如下的抽象类：

```
Public abstract class CloneableType
{
  // 只有派生类型才能支持“多态接口”。在其他层次结构中的类不能访问这个抽象成员
  public abstract object Clone();
}
```

按照这个定义，只有扩展了CloneableType的成员才能支持Clone()方法。如果我们创建的新的集合类没有扩展这个基类，就不能获取这个多态接口。你可能还记得C#不支持类的多重继承。因此，如果想创建既是Car又是CloneableType的MiniVan，下面的代码是行不通的：

```
// 不行！C#不允许类的多重继承
public class MiniVan : Car, CloneableType
{
}
```

你可能也想到了，接口类型就是来解决这个问题的。在定义了接口之后，它就可以被任何层次结构、任何命名空间或任何程序集（由任何.NET编程语言写的）中的任何类或结构来实现。这样的话，接口就有较高的多态性。考虑定义在System命名空间中名为ICloneable的标准.NET接口，它定义了一个名为Clone()的方法：

```
public interface ICloneable
{
  object Clone();
}
```

① Robert Martin在《敏捷软件开发（C#版）》（人民邮电出版社，2008）一书中指出，因为在重构时接口和类常常相互转换，所以这一命名约定在系统框架之外普通程序的开发中并不合理。请读者注意。——编者注

如果研究.NET Framework 4.5 SDK文档的话，就会发现非常多看似无关的类型（System.Array、System.Data.SqlClient.SqlConnection、System.OperatingSystem、System.String等）都实现了这个接口。尽管这些类型不具有相同的父类（除了System.Object之外），我们可以通过ICloneable接口类型把它们当成多态处理。

例如，如果我们有一个含ICloneable接口参数的方法CloneMe()，我们就可以把任何实现这个接口的对象传给这个方法。考虑如下定义在ICloneExample控制台应用程序中的简单Program类：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** A First Look at Interfaces *****\n");

    // 所有这些类都支持ICloneable接口
    string myStr = "Hello";
    OperatingSystem unixOS = new OperatingSystem(PlatformID.Unix, new Version());
    System.Data.SqlClient.SqlConnection sqlCnn =
      new System.Data.SqlClient.SqlConnection();

    // 因此，它们就可以传入接口ICloneable的方法
    CloneMe(myStr);
    CloneMe(unixOS);
    CloneMe(sqlCnn);
    Console.ReadLine();
  }

  private static void CloneMe(ICloneable c)
  {
    // 克隆我们获得的并输出名字
    object theClone = c.Clone();
    Console.WriteLine("Your clone is a: {0}",
      theClone.GetType().Name);
  }
}
```

如果我们运行这个应用程序，就会发现每一个类的完整名都被输出到了控制台。这是通过从System.Object中继承的GetType()方法来实现的。第15章会讲到，这个方法以及.NET反射服务会让我们理解运行时任何类型的组成。任何情况下，前一个项目的输出都如下所示：

```
***** A First Look at Interfaces *****

Your clone is a: String
Your clone is a: OperatingSystem
Your clone is a: SqlConnection
```

传统抽象基类的另外一个限制就是每一个派生类型必须处理这一组抽象成员并且提供实现。为了演示这个问题，回忆一下我们在第6章中定义的图形层次结构。假设我们在Shape基类中新定义了一个叫GetNumberOfPoints()的抽象方法，它允许派生类型返回渲染图形所需的顶点数：

```
abstract class Shape
{
...
```

```
    // 每一个派生类型都必须支持这个方法
    public abstract byte GetNumberOfPoints();
  }
```

显然，只有Hexagon类型才拥有顶点。然而，这样更新之后，所有派生类型（Circle、Hexagon以及ThreeDCircle）现在都必须提供这个方法完整的实现，即使这么做没有什么意义。同样，接口类型提供了解决方案。如果我们定义了一个接口来表示“有顶点”这个行为，我们就可以把它插到Hexagon类型中，Circle和ThreeDCircle则不受影响。

源代码　ICloneableExample项目的源代码位于Chapter 8子目录下。

8.2　定义自定义接口

现在我们较好地理解了接口类型的总体作用，那么就让我们来看一个定义和实现自定义接口的示例。首先，创建一个全新的控制台应用程序CustomInterface。使用Project→Add Existing Item菜单项，插入在第6章Shapes示例中创建的包含我们图形类型定义的文件（本书解决方案代码中的Shape.cs）。然后，将定义图形相关的类型的命名空间重命名为CustomInterface（只是为了避免将命名空间定义导入到我们的新项目中）：

```
namespace CustomInterface
{
  // 在这里是我们之前的图形类型
}
```

现在，使用Project→Add New Item菜单项插入一个新接口到我们的项目中并起名为IPointy，如图8-1所示。

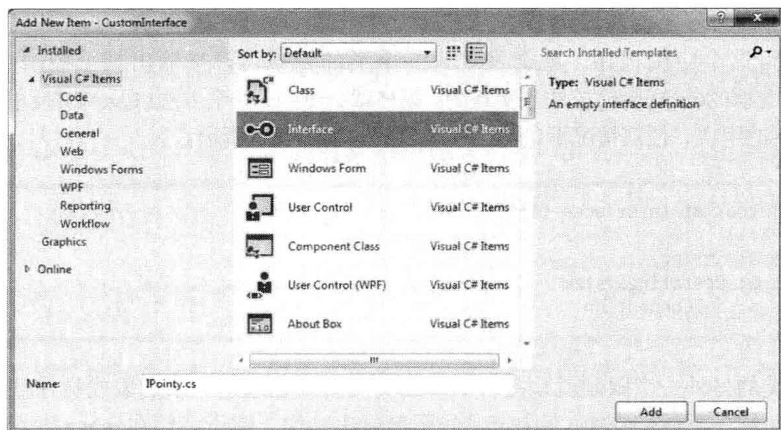


图8-1　和类相似，接口可以定义在任何*.cs文件中

从语法级别来说，接口使用C# interface关键字来定义。和类不一样的是，接口不指定基类（即使是System.Object；然而，正如你在本章将看到的那样，接口可以指定基接口）。而且接口的成员也不指

定访问修饰符（因为所有接口成员都是隐式公共的和抽象的）。这里是一个使用C#定义的自定义接口：

```
// 这个接口定义了“具有顶点”的行为
public interface IPointy
{
  // 隐式公共的和抽象的
  byte GetNumberOfPoints();
}
```

记住，当我们定义接口成员时，不需要为这个成员定义实现作用域。接口是纯粹的协议，因此也不会定义实现（留给支持的类或结构）。因此，如下版本的IPointy会导致各种编译器错误：

```
// 内有大量错误
public interface IPointy
{
  // 错误！接口不能有字段
  public int numbOfPoints;

  // 错误！接口不能有构造函数
  public IPointy() { numbOfPoints = 0;};

  // 错误！接口不能提供实现
  byte GetNumberOfPoints() { return numbOfPoints; }
```

不管怎么样，原始的IPointy接口定义了一个方法。然而，.NET接口还可以定义许多属性协议。例如，我们可以使用只读属性而不是其他的访问方法来创建IPointy接口：

```
// 这个pointy表现为一个只读属性
public interface IPointy
{
  // 在接口中的读写属性差不多是retType PropName { get; set; }

  // 而接口中的只写属性是retType PropName { set; }

  byte Points { get; }
}
```

说明 接口类型还可以包含事件（见第10章）以及索引器（见第11章）定义。

接口类型就其本身而言没什么用，因为它们只是抽象成员的集合。例如，我们不能像类和结构一样分配接口类型：

```
// 分配接口类型是不合法的
static void Main(string[] args)
{
  IPointy p = new IPointy();  // 编译器错误
}
```

除非被类或结构实现，否则接口没有什么用。在这里，IPointy是一个表示“有顶点”这一行为的接口。原因很简单，图形层次结构中的一些类有顶点（如Hexagon），而其他一些则没有（比如Circle）。

8.3 实现接口

如果类（或结构）选择通过支持接口来扩展功能，就需要在其类型定义中使用逗号分隔的列表。要知道直接基类必须是冒号操作符后的第一个项。如果类类型从System.Object直接继承，我们完全可

以只在列表中提供类支持的接口，因为如果没有特别指明，C#编译器会从System.Object扩展我们的类型。由于结构总是从System.ValueType继承（完整细节见第4章），只需要在结构定义后直接列出每一个接口就行了。考虑如下的示例：

```
// 这个类派生自System.Object并且实现一个接口
public class Pencil : IPointy
{...}

// 这个类也派生自System.Object并且实现一个接口
public class SwitchBlade : object, IPointy
{...}

// 这个类派生自一个自定义基类并且实现一个接口
public class Fork : Utensil, IPointy
{...}

// 这个结构隐式派生自System.ValueType并且实现两个接口
public struct Arrow : ICloneable, IPointy
{...}
```

要理解，实现接口是一个“要么全要要么全不要”的命题，也就是说支持类型无法选择实现哪些成员。由于IPointy接口定义了一个只读属性，因此问题不大。然而，如果我们要实现一个定义10个成员的接口（比如前面说的IDbConnection接口），这个类型就需要充实10个抽象实体的细节。

例如，插入一个叫Triangle的新类类型，它是一个Shape，并且支持IPointy。注意只读属性Points的实现只返回正确的顶点数（3）：

```
// 名为Triangle（三角形）的新的Shape派生类型
class Triangle : Shape, IPointy
{
  public Triangle() { }
  public Triangle(string name) : base(name) { }
  public override void Draw()
  { Console.WriteLine("Drawing {0} the Triangle", PetName); }

  // IPointy实现
  public byte Points
  {
    get { return 3; }
  }
}
```

现在，更新既有的Hexagon类型来支持IPointy接口类型：

```
// Hexagon现在实现IPointy
class Hexagon : Shape, IPointy
{
  public Hexagon(){ }
  public Hexagon(string name) : base(name){ }
  public override void Draw()
  { Console.WriteLine("Drawing {0}  the Hexagon", PetName); }

  // IPointy实现
  public byte Points
  {
    get { return 6; }
  }
}
```

现在来总结一下，图8-2显示的Visual Studio类结构图使用流行的“棒棒糖”符号描述了与IPointy兼容的类。注意，Circle和ThreeDCircle没有实现IPointy，因为这个行为对这些特殊类没有意义。

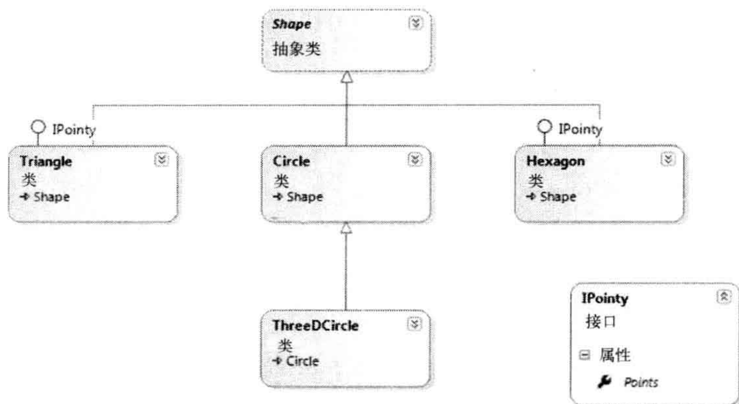


图8-2 Shape层次结构（包含接口）

说明 通过右击接口图标并选择Collapse或Expand，可以分别显示或隐藏类设计器上的接口名称。

8.4 在对象级别调用接口成员

现在已经有了一组支持IPointy接口的类，接下来的问题就是如何使用这些新功能。与给定接口功能最直接的交互方式就是直接在对象级别调用方法（所提供的接口成员不是显式实现的，在8.9节中将有详解）。例如，考虑下面的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");

  // 调用IPointy定义的Points属性
  Hexagon hex = new Hexagon();
  Console.WriteLine("Points: {0}", hex.Points);
  Console.ReadLine();
}
```

由于读者清楚六边形（Hexagon）类型已经实现了该接口，有了Points属性，所以在本例中这样做没有任何问题。但在其他情况下，读者可能无法在编译时判断指定类型支持哪个接口。例如，假定读者有一个包含50个Shape兼容类型的数组，其中仅有部分数组支持IPointy接口。很明显，如果试图在没有实现IPointy接口的类型中调用Points属性，将收到编译时错误。接下来的问题就是：如何才能动态判断一个类型支持哪些接口呢？

在运行时判断一个类型是否支持一个指定接口的一种方式是使用显式强制转换。如果这个类型不支持被请求的接口，将收到一个无效转换异常（InvalidCastException）。使用结构化异常处理妥善处置这种可能的异常，例如：

```
static void Main(string[] args)
{
...
  // 捕获可能发生的InvalidCastException异常
  Circle c = new Circle("Lisa");
  IPointy itfPt = null;
  try
  {
    itfPt = (IPointy)c;
    Console.WriteLine(itfPt.Points);
  }
  catch (InvalidCastException e)
  {
    Console.WriteLine(e.Message);
  }

  Console.ReadLine();
}
```

使用try/catch逻辑并非是最好的解决方法，在首次调用该接口成员之前判断其支持哪个接口更加理想。下面介绍两种实现方式。

8.4.1　获取接口引用：as关键字

判断一个指定类型是否支持一个接口的第二种方式就是使用第6章介绍的as关键字。如果该对象可被视为一个指定的接口，你可以在使用该关键字的语句中得到指向该对象接口的引用；否则，将返回一个值为null的空引用。因此，首先要检查null值：

```
static void Main(string[] args)
{
...
  // 我们能将六角形hex2视为实现了IPointy接口吗
  Hexagon hex2 = new Hexagon("Peter");
  IPointy itfPt2 = hex2 as IPointy;

  if(itfPt2 != null)
    Console.WriteLine("Points: {0}", itfPt2.Points);
  else
    Console.WriteLine("OOPS! Not pointy...");
  Console.ReadLine();
}
```

请注意，当使用as关键字的时候，无需使用try/catch逻辑。如果引用非空，说明调用的是一个正确的接口引用。

8.4.2　获取接口引用：is关键字

还可以通过使用is关键字（第6章介绍过）来检查是否实现一个接口。如果要考查的对象与指定接口不符，将返回false值。反之，如果该类型与指定接口相符，就可以安全地调用这些成员，而不必使用try/catch逻辑。

假定更新了Shape类型的数组，使其中部分成员实现了IPointy接口。注意，下面的Main()方法显示了如何使用is关键字判断数组中哪些项支持该接口：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");

  //生成Shape数组
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle("Joe"), new Circle("JoJo")} ;

  for(int i = 0; i < myShapes.Length; i++)
  {
    // 回调Shape基类定义一个抽象的Draw()成员，由此所有Shape都知道如何绘制自己
    myShapes[i].Draw();

    // 哪些是有棱角的
    if(myShapes[i] is IPointy)
      Console.WriteLine("-> Points: {0}", ((IPointy) myShapes[i]).Points);
    else
      Console.WriteLine("-> {0}\'s not pointy!", myShapes[i].PetName);
    Console.WriteLine();
  }
  Console.ReadLine();
}
```

输出结果如下所示：

```
***** Fun with Interfaces *****

Drawing NoName the Hexagon
-> Points: 6

Drawing NoName the Circle
-> NoName's not pointy!

Drawing Joe the Triangle
-> Points: 3

Drawing JoJo the Circle
-> JoJo's not pointy!
```

8.5　接口作为参数

既然接口是有效的.NET类型，读者可以构造将接口作为参数的方法，和本章前面的CloneMe()一样。对于当前的示例，假定已经定义了另一个名为IDraw3D的接口：

```
// 模拟能以绝佳3D效果呈现一个类型的能力
public interface IDraw3D
{
  void Draw3D();
}
```

接下来，假定3种图形中的2种（ThreeDCircle与Hexagon）已经被设定为支持这种新的行为：

```
// Circle支持IDraw3D接口
class ThreeDCircle : Circle, IDraw3D
{
...
```

```
  public void Draw3D()
  {  Console.WriteLine("Drawing Circle in 3D!"); }
}

// Hexagon支持IPointy与IDraw3D接口
class Hexagon : Shape, IPointy, IDraw3D
{
...
  public void Draw3D()
  {  Console.WriteLine("Drawing Hexagon in 3D!"); }
}
```

图8-3所示为新的Visual Studio类图。

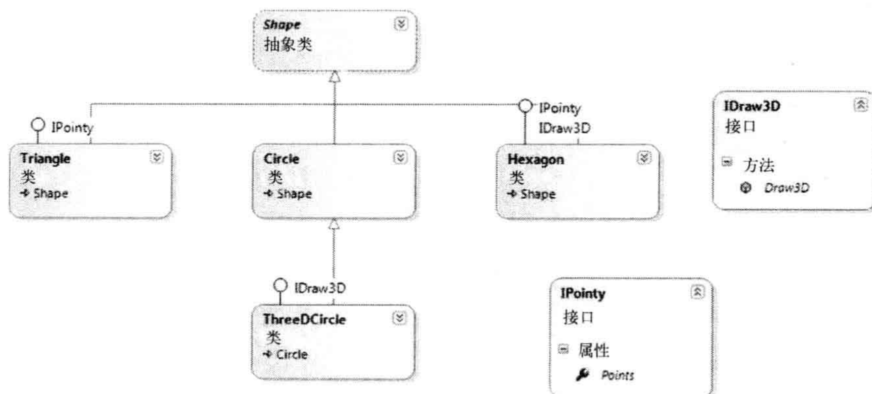


图8-3　更新的Shape层次结构

如果读者现在定义一个将IDraw3D接口作为参数的方法，将能有效传递任何实现IDraw3D接口的对象（如果读者试图传进一个不支持该接口的类型，将收到编译时错误）。思考下面这个在Program类中定义的方法：

```
// 绘制任何支持IDraw3D接口的类型
static void DrawIn3D(IDraw3D itf3d)
{
  Console.WriteLine("-> Drawing IDraw3D compatible type");
  itf3d.Draw3D();
}
```

可以测试Shape数组中的项是否支持接口，如果支持，就将其传入DrawIn3D()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle(), new Circle("JoJo") } ;
  for(int i = 0; i < myShapes.Length; i++)
  {
    ...
    // 支持绘制为3D吗
    if(myShapes[i] is IDraw3D)
      DrawIn3D((IDraw3D)myShapes[i]);
  }
}
```

下面是修改后的应用程序的输出结果。注意只有Hexagon对象以3D形式输出， Shape数组的其他成员都没有实现IDraw3D接口：

```
***** Fun with Interfaces *****

Drawing NoName the Hexagon
-> Points: 6
-> Drawing IDraw3D compatible type
Drawing Hexagon in 3D!

Drawing NoName the Circle
-> NoName's not pointy!

Drawing Joe the Triangle
-> Points: 3

Drawing JoJo the Circle
-> JoJo's not pointy!
```

8.6 接口作为返回值

接口也可以被用来作为方法的返回值。例如，读者可以写一个接受Shape对象数组作为参数、返回支持IPointy的第一项的引用的方法：

```
// 这个方法返回实现IPointy的数组中的第一个对象
static IPointy FindFirstPointyShape(Shape[] shapes)
{
  foreach (Shape s in shapes)
  {
    if (s is IPointy)
      return s as IPointy;
  }
  return null;
}
```

我们可以与这个方法按如下代码进行交互：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  // 构建Shape数组
  Shape[] myShapes = { new Hexagon(), new Circle(),
                       new Triangle("Joe"), new Circle("JoJo")};

  // 获取第一个pointy项
  // 安全起见，在使用前最好检查firstPointyItem是否为null
  IPointy firstPointyItem = FindFirstPointyShape(myShapes);
  Console.WriteLine("The item has {0} points", firstPointyItem.Points);
...
}
```

8.7 接口类型数组

要理解的是，同样的接口即使不在同一个类层次结构，也没有除System.Object以外的公共父类，

也可以由多个类型实现，这可以派生出许多非常强大的编程结构。例如，假设我们要在当前项目中开发三个全新的类类型来对厨具（通过Knife和Fork类）和园艺设备（PitchFork，指干草叉）建模。如图8-4所示。

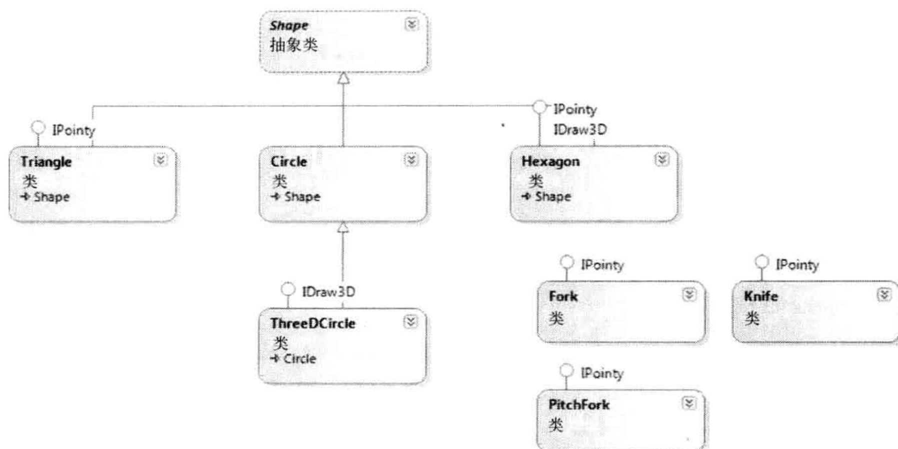


图8-4　接口可以插入到类层次结构任何部分的类型中

如果已经定义了PitchFork、Fork和Knife类型，那么现在可以定义一个支持IPointy接口的对象数组。既然这些成员都支持同样的接口，因此可以抛开类层次结构的全部差异性，通过数组进行迭代并将每个对象视为支持IPointy接口的对象：

```
static void Main(string[] args)
{
...
  // 这个数组仅仅包含实现了IPointy接口的类型
  IPointy[] myPointyObjects = {new Hexagon(), new Knife(),
    new Triangle(), new Fork(), new PitchFork()};

  foreach(IPointy i in myPointyObjects)
    Console.WriteLine("Object has {0} points.", i.Points);
  Console.ReadLine();
}
```

下面强调一下这个示例的重要性，请记住：如果你有一个给定接口的数组，那么这个数组可以包含实现了该接口的任何类或者结构。

源代码　CustomInterface项目的源代码位于Chapter 8的子目录下。

8.8 使用 Visual Studio 实现接口

尽管基于接口编程是非常强大的编程技术，实现接口却需要大量的代码键入。由于接口是一组具名的抽象成员，每个支持该行为的类型中的每个接口方法，都需要键入定义和实现。

正如我们所期待的，Visual Studio支持多个工具来减少实现接口的工作任务量。通过简单的测试，将最后一个类插入到当前项目PointyTestClass中。当实现IPointy接口（或者其他接口）时，注意当完成键入接口名称或在代码窗口中将鼠标光标移到接口名称上时，第一个字母是带下划线的（正式名称为智能标签）。单击智能标签，一个用来实现接口的下拉列表将显示出来，如图8-5所示。

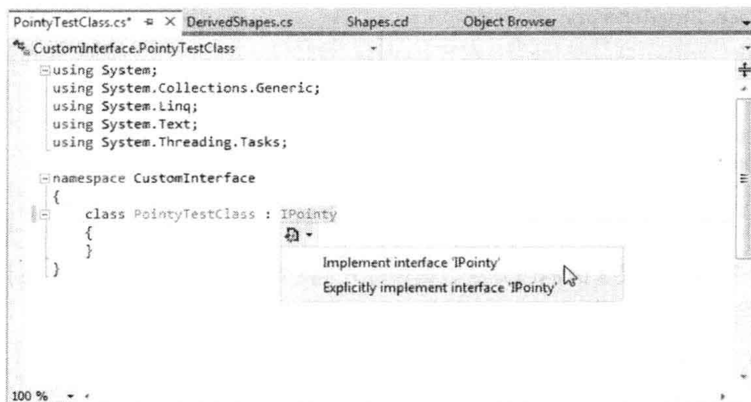


图8-5　使用Visual Studio实现接口

注意图中有两个选项，下一节将详细介绍第2个选项（显式接口实现）。选定第一个选项后，将看到Visual Studio在一个已命名的代码区块中内置生成了存根代码供读者修改（注意，其中默认实现抛出System.NotImplementedException异常，它显然能被删除掉）。

```
namespace CustomInterface
{
  class PointyTestClass : IPointy
  {
    public byte Points
    {
      get { throw new NotImplementedException(); }
    }
  }
}
```

说明　Visual Studio还支持提取Refactor菜单中的接口重构。这允许我们从既有类定义中提取新的接口定义。例如，你可能在编写一个类的中途发现可以将其行为归纳为一个接口（这样就可以提供另一种实现）。

8.9　显式接口实现

之前说过，一个类或结构可以实现许多接口。因此，我们可能实现包含重复命名成员的接口，所以就需要处理命名冲突。为了演示解决这个问题的各种方式，新建一个叫InterfaceNameClash的控制台应用程序。现在设计3个自定义接口来表示实现类型呈现自身输出的各种位置：

```
// 绘制到表单上
public interface IDrawToForm
{
  void Draw();
}

// 绘制到内存中
public interface IDrawToMemory
{
  void Draw();
}

// 呈现到打印机
public interface IDrawToPrinter
{
  void Draw();
}
```

注意，每一个接口都定义了Draw()方法，其名称相同（碰巧都没有参数）。如果我们现在希望一个名为Octagon的类类型支持这些接口中的每一个，编译器会允许如下的定义：

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  public void Draw()
  {
    // 共享绘制逻辑
    Console.WriteLine("Drawing the Octagon...");
  }
}
```

尽管这段代码可以通过编译，你可能也认为我们会有问题。简单来说，如果提供一个Draw()方法实现，我们就不能从Octagon对象根据某个接口采取一系列行动。例如，下面的代码会调用相同的Draw()方法，而不管我们获取到哪个接口：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
  // 所有这些调用都会调用相同的Draw()方法
  Octagon oct = new Octagon();

  IDrawToForm itfForm = (IDrawToForm)oct;
  itfForm.Draw();

  IDrawToPrinter itfPriner = (IDrawToPrinter)oct;
  itfPriner.Draw();

  IDrawToMemory itfMemory = (IDrawToMemory)oct;
  itfMemory.Draw();

  Console.ReadLine();
}
```

显然，把图像呈现到窗体的代码和把图像呈现到网络打印机或内存中某个区域的代码不太一样。如果要实现具有相同成员的接口，可以使用*显式接口实现语法*来解决这种命名冲突。考虑如下对Octagon类型的更新：

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
```

```
    // 对某个接口显式绑定Draw()
    void IDrawToForm.Draw()
    {
      Console.WriteLine("Drawing to form...");
    }
    void IDrawToMemory.Draw()
    {
      Console.WriteLine("Drawing to memory...");
    }
    void IDrawToPrinter.Draw()
    {
      Console.WriteLine("Drawing to a printer...");
    }
  }
```

我们可以看到，如果显式实现接口成员的话，大致模式可以归结为：

```
returnType InterfaceName.MethodName(params){}
```

如果使用这个语法，我们只需要提供一个访问修饰符，显式实现的成员是自动私有的。例如，下面的语法是不合法的：

```
// 错误！没有访问修饰符
public void IDrawToForm.Draw()
{
  Console.WriteLine("Drawing to form...");
}
```

由于显示实现的成员总是隐式私有的，这些成员在对象级别就不可用。其实，如果我们把点操作符应用到Octagon类型，就会发现智能感知不会显示我们的Draw()方法。可以想象，我们必须使用显式转换来访问需要的功能。例如：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interface Name Clashes *****\n");
  Octagon oct = new Octagon();

  // 现在必须使用转换来访问Draw()成员
  IDrawToForm itfForm = (IDrawToForm)oct;
  itfForm.Draw();

  // 如果以后不需要接口变量，可以简化成这个形式
  ((IDrawToPrinter)oct).Draw();

  // 也可以使用“as”关键字
  if(oct is IDrawToMemory)
    ((IDrawToMemory)oct).Draw();

  Console.ReadLine();
}
```

虽然这个语法对解决命名冲突很有用，但是如果希望从对象级别隐藏“高级”成员的话，我们也可以使用显式接口实现。这样，如果对象用户使用点操作符的话，他就只能看到类型所有功能的子集。然而，那些需要更多高级行为的人可以通过显式转换提取需要的接口。

源代码 InterfaceNameClash项目的源代码位于Chapter 8子目录下。

8.10 设计接口层次结构

接口可以组织成接口层次结构。和类层次结构相似，如果接口扩展了既有接口，它就继承了父类定义的抽象成员。当然，和基于类的继承不同的是，派生接口不会继承真正的实现，而只是通过额外的抽象成员扩展了其自身的定义。

如果希望扩展既有接口功能又不变动既有代码，接口层次结构就会很有用。为了举例说明，让我们新建一个控制台应用程序InterfaceHierarchy。现在，让我们重新设计之前的一组与呈现相关的接口，这样IDrawable就是家族树的根：

```
public interface IDrawable
{
  void Draw();
}
```

由于IDrawable定义了基本绘制行为，我们现在就可以创建派生接口来扩展以修改后的格式呈现的能力，例如：

```
public interface IAdvancedDraw : IDrawable
{
  void DrawInBoundingBox(int top, int left, int bottom, int right);
  void DrawUpsideDown();
}
```

有了这样的设计，如果一个类实现IAdvancedDraw，我们现在就必须实现在继承链上定义的每一个成员（更准确地说是Draw()、DrawInBoundingBox()和DrawUpsideDown()方法）：

```
public class BitmapImage : IAdvancedDraw
{
  public void Draw()
  {
    Console.WriteLine("Drawing...");
  }

  public void DrawInBoundingBox(int top, int left, int bottom, int right)
  {
    Console.WriteLine("Drawing in a box...");
  }

  public void DrawUpsideDown()
  {
    Console.WriteLine("Drawing upside down!");
  }
}
```

现在，使用BitmapImage时，可以在对象级别上调用每一个方法（因为它们都是公有的），也可以通过显式转换提取每一个支持接口的引用：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Interface Hierarchy *****");
```

```
    // 从对象级别调用
    BitmapImage myBitmap = new BitmapImage();
    myBitmap.Draw();
    myBitmap.DrawInBoundingBox(10, 10, 100, 150);
    myBitmap.DrawUpsideDown();

    // 显式获取IAdvancedDraw
    IAdvancedDraw iAdvDraw = myBitmap as IAdvancedDraw;
    If(iAdvDraw ! = null)
      iAdvDraw.DrawUpsideDown();

    Console.ReadLine();
  }
```

源代码 InterfaceHierarchy项目的源代码位于Chapter 8子目录下。

接口类型的多重继承

和类类型不同，一个接口可以扩展多个基接口。这就允许我们设计非常强大、非常灵活的抽象。新建一个控制台应用程序MIInterfaceHierarchy。这是一套全新的接口，模拟了各种呈现以及图形相关的抽象。注意IShape接口扩展了IDrawable和IPrintable：

```
// 接口可以是多重继承的
interface IDrawable
{
   void Draw();
}

interface IPrintable
{
   void Print();
   void Draw(); // <-- 注意，可能导致命名冲突
}

// 多重接口继承。没有问题
interface IShape : IDrawable, IPrintable
{
   int GetNumberOfSides();
}
```

图8-6说明了当前的接口层次结构。

现在，关键问题就是如果我们有一个类支持IShape，需要实现多少方法呢？回答是：看情况。如果希望提供Draw()的简单实现，只需要提供3个成员，如下面的Rectangle类型所示：

```
class Rectangle : IShape
{
  public int GetNumberOfSides()
  { return 4; }

  public void Draw()
  { Console.WriteLine("Drawing..."); }

  public void Print()
  { Console.WriteLine("Prining..."); }
}
```

8

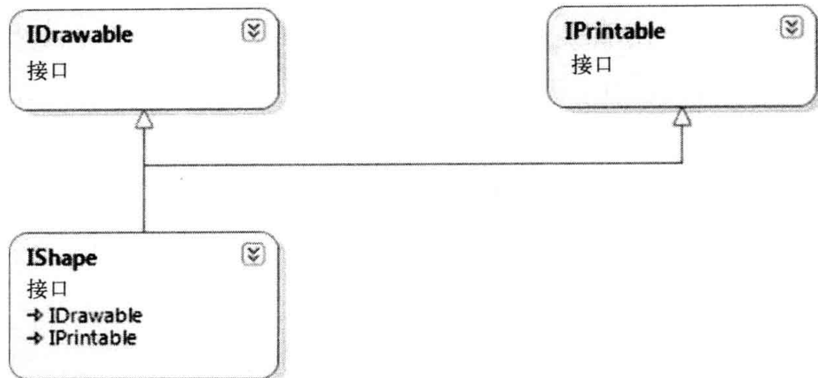


图8-6 和类不一样，接口可以扩展多个接口类型

如果我们更愿意对每一个Draw()方法提供特定实现（这里应该比较有意义的），就可以使用显式接口实现解决命名冲突，如下面的Square类型所示：

```
class Square : IShape
{
  // 使用显式实现来处理成员命名冲突
  void IPrintable.Draw()
  {
  { // 绘制到打印机上
  }
  void IDrawable.Draw()
  {
  { // 绘制到屏幕上
  }
  public void Print()
  {
  // 打印
  }
  public int GetNumberOfSides()
  { return 4; }
}
```

至此，你应该比较熟悉使用C#语法来定义和实现自定义接口的过程了。坦白地说，基于接口的编程不是那么容易理解的，如果你确实还有点糊涂，这完全是正常的。

然而，需要知道，接口是.NET框架的一个基本方面。不管我们在开发什么类型的应用程序（基于Web、桌面GUI还是数据访问类库等），总会要用到接口。最后来总结一下，接口在以下情况下特别有用：

- 只有一个层次结构，但是只有一个派生类型的子集支持公共行为；
- 需要构建的公共行为跨多个层次结构，而且除了System.Object以外，没有其他公共父类。

既然我们已经研究了构建和实现自定义接口的细节，本章剩余部分会研究在.NET基础类库中包含的许多预定义接口。

源代码 MIInterfaceHierarchy项目的源代码位于Chapter 8子目录下。

8.11 构建可枚举类型（IEnumerable和IEnumerator）

为了开始对实现既有.NET接口的研究，让我们先看一下IEnumerable和IEnumerator的作用。回忆一下，C#支持关键字foreach，允许我们遍历任何数组类型的内容：

```
// 遍历数组的项
int[] myArrayOfInts = {10, 20, 30, 40};

foreach(int i in myArrayOfInts)
{
  Console.WriteLine(i);
}
```

虽然看上去只有数组类型才可以使用这个结构，其实任何支持GetEnumerator()方法的类型都可以通过foreach结构进行运算。举例说明，首先新建一个叫CustomEnumerator的控制台应用程序。然后，增加在第7章SimpleException示例中定义的Car.cs和Radio.cs文件（通过Project→Add Existing Item菜单项）。

说明 你可能希望将包含Car和Radio类型的命名空间重命名为CustomEnumerator，这样就可以避免将CustomException命名空间导入到新项目中。

8

现在，插入一个新类Garage，它在System.Array中保存了一组Car类型：

```
// Garage包含一组Car对象
public class Garage
{
  private Car[] carArray = new Car[4];

  // 启动时填充一些Car对象
  public Garage()
  {
    carArray[0] = new Car("Rusty", 30);
    carArray[1] = new Car("Clunker", 55);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
  }
}
```

理想情况下，与数据值数组一样，使用foreach构造迭代Garage对象中的每个子项比较方便：

```
// 这看起来好像是可行的
public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with IEnumerable / IEnumerator *****\n");
    Garage carLot = new Garage();

    // 移交集合中的每个Car对象吗
    foreach (Car c in carLot)
    {
      Console.WriteLine("{0} is going {1} MPH",
        c.PetName, c.CurrentSpeed);
```

```
        }
        Console.ReadLine();
      }
    }
```

让人沮丧的是，编译器通知我们Garage类没有实现名为GetEnumerator()的方法。这个方法是由隐藏在System.Collections命名空间中的IEnumerable接口定义的。

说明　下一章将介绍泛型的作用以及System.Collections.Generic命名空间。你将看到，该命名空间包含IEnumerable/IEnumerator的泛型版本，它为子对象的迭代提供类型更加安全的方式。

支持这种行为的类或结构实际上是在宣告它们向调用者（如foreach关键字本身）公开了所包含的子项，下面是标准的.NET接口定义：

```
// 这个接口告知调用方对象的子项可以枚举
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

可以看到，GetEnumerator()方法返回一个对另一个接口System.Collections.IEnumerator的引用。这个接口提供了基础设施，调用方可以用来移动IEnumerable兼容容器包含的内部对象：

```
// 这个接口允许调用方获取一个容器的子项
public interface IEnumerator
{
  bool MoveNext ();          // 将光标的内部位置向前移动
  object Current { get;}     // 获取当前的项（只读属性）
  void Reset ();             // 将光标重置到第一个成员前面
}
```

如果想修改Garage类型使之支持这些接口，可以手工实现每个方法，这需要花费不少精力。虽然自己开发GetEnumerator()、MoveNext()、Current和Reset()也没问题，但有一个更简单的办法。因为System.Array类型和其他许多类型已经实现了IEnumerable和IEnumerator接口，你可以简单地将请求委托到System.Array，如下所示：

```
using System.Collections;
...
public class Garage : IEnumerable
{
  // System.Array已经实现了IEnumerator
  private Car[] carArray = new Car[4];

  public Garage()
  {
    carArray[0] = new Car("FeeFee", 200, 0);
    carArray[1] = new Car("Clunker", 80, 0);
    carArray[2] = new Car("Zippy", 30, 0);
    carArray[3] = new Car("Fred", 30, 0);
  }

  public IEnumerator GetEnumerator()
  {
    // 返回数组对象的IEnumerator
```

```
        return carArray.GetEnumerator();
    }
}
```

修改Garage类型之后，就可以在C# foreach结构中安全使用该类型了。除此之外，GetEnumerator()被定义为公开的，对象用户可以与IEnumerator类型交互：

```
// 手动与IEnumerator协作
IEnumerator i = carLot.GetEnumerator();
i.MoveNext();
Car myCar = (Car)i.Current;
Console.WriteLine("{0} is going {1} MPH", myCar.PetName, myCar.CurrentSpeed);
```

如果希望在对象级隐藏IEnumerable的功能，只需要使用显式接口实现就行了：

```
IEnumerator IEnumerable.GetEnumerator()
{
  // 返回数组对象的IEnumerator
  return carArray.GetEnumerator();
}
```

这样的话，对象用户就不能找到Garage的GetEnumerator()方法，而foreach结构会在必要的时候在背后获得接口。

源代码 CustomEnumerator项目的源代码位于Chapter 8子目录下。

8

8.11.1 用yield关键字构建迭代器方法

在以前，如果我们希望构建支持foreach枚举的自定义集合（如Garage），只能实现IEnumerable接口（可能还有IEnumerator接口）。然而，还可以通过迭代器来构建使用foreach循环的类型。

简单来说，迭代器就是这样一个成员方法，它指定了容器内部项被foreach处理时该如何返回。虽然迭代器方法还必须命名为GetEnumerator()，返回值还是必须为IEnumerator类型，但自定义类不需要实现原来那些接口了。

为了演示，新建一个Console Application项目CustomEnumeratorWithYield，并添加前面示例中的Car、Radio和Garage类型（如果愿意的话，可以将命名空间的名称改为当前项目）。现在，对当前的Garage类型做如下改进：

```
public class Garage:IEnumerator
{
  private Car[] carArray = new Car[4];
  ...
  // 迭代器方法
  public IEnumerator GetEnumerator()
  {
    foreach (Car c in carArray)
    {
      yield return c;
    }
  }
}
```

注意，这个GetEnumerator()的实现使用内部foreach逻辑迭代每个子项，使用新的yield return语法向调用方返回每个Car对象。yield关键字用来向调用方的foreach结构指定返回值。当到达yield return语句后，当前位置被存储下来，下次调用迭代器时会从这个位置开始执行。

迭代器方法不一定要通过foreach关键字来返回内容。我们也可以使用如下代码定义迭代器方法：

```
public IEnumerator GetEnumerator()
{
  yield return carArray[0];
  yield return carArray[1];
  yield return carArray[2];
  yield return carArray[3];
}
```

在这个实现中，注意GetEnumerator()方法显式返回新的值给调用者。虽然对于这个示例来说意义不是很大，因为如果我们为carArray成员变量增加更多对象的话，GetEnumerator()方法就不会同步。但是，如果我们希望方法返回能被foreach语法处理的局部数据，这个语法就很有用。

8.11.2　构建命名迭代器

还有有趣的一点是，yield关键字从技术上说可以结合任何方法一起使用，无论方法名是什么。这些方法（技术上称为*命名迭代器*）独特之处在于可以接受许多参数。如果构建命名迭代器的话，需要知道这些方法会返回IEnumerable接口，而不是预计的IEnumerator兼容类型。例如，我们可以为Garage类型增加如下方法：

```
public IEnumerable GetTheCars(bool ReturnReversed)
{
  // 逆序返回项
  if (ReturnReversed)
  {
    for (int i = carArray.Length; i != 0; i--)
    {
      yield return carArray[i-1];
    }
  }
  else
  {
    // 按顺序返回数组中的项
    foreach (Car c in carArray)
    {
      yield return c;
    }
  }
}
```

注意，我们的新方法允许调用者以正序和逆序（如果传入的参数值为true）来获取子项。我们可以按如下所示的代码和新方法进行交互：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with the Yield Keyword *****\n");
  Garage carLot = new Garage();

  // 使用GetEnumerator()获取项
  foreach (Car c in carLot)
```

```
    {
      Console.WriteLine("{0} is going {1} MPH",
        c.PetName, c.CurrentSpeed);
    }

    Console.WriteLine();

    // 使用命名迭代器来获取项（逆序）
    foreach (Car c in carLot.GetTheCars(true))
    {
      Console.WriteLine("{0} is going {1} MPH",
        c.PetName, c.CurrentSpeed);
    }
    Console.ReadLine();
  }
```

命名迭代器是很有用的结构，因为一个自定义容器可以定义多重方式来请求返回的集。

那么，总结一下可枚举对象的构建吧。记住，如果自定义类型要和C#的foreach关键字一起使用的话，容器就需要定义一个名为GetEnumerator()的方法，它由IEnumerator接口类型来定制。通常，这个方法的实现只是交给保存子对象的内部成员，然而，我们也可以使用yield return语法来提供多个“命名迭代器”方法。

源代码 CustomEnumeratorWithYield项目的源代码位于Chapter 8子目录下。

8

8.12 构建可克隆的对象（ICloneable）

回忆一下第6章，System.Object定义了一个名为MemberwiseClone()的成员。这个方法用来获取当前对象的一份浅复制。因为它是受保护的，对象用户不会直接调用这个方法，而一个对象可能在克隆过程中自己调用这个方法。为便于说明，创建一个名为CloneablePoint的新控制台应用程序项目，其中定义了一个名为Point的类：

```
// 一个名为Point的类
public class Point
{
  public int X {get; set;}
  public int Y {get; set;}

  public Point(int xPos, int yPos) { X = xPos; Y = yPos;}
  public Point(){}

  // 重写Object.ToString()
  public override string ToString()
  {  return string.Format("X = {0}; Y = {1}", X, Y ); }
}
```

在第4章中你已经知道了引用类型与值类型，注意如果给一个引用变量分配另一个引用变量，将有两个引用指向内存中的同一个对象。下面的赋值操作将导致两个引用指向堆上的同一个Point对象，通过任何一个引用都能修改堆上的同一对象：

```
static void Main(string[] args)
{
```

```
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    // 指向同一对象的两个引用
    Point p1 = new Point(50, 50);
    Point p2 = p1;
    p2.X = 0;
    Console.WriteLine(p1);
    Console.WriteLine(p2);
    Console.ReadLine();
  }
```

如果读者想使自己的自定义类型支持向调用方返回自身同样副本的能力，需要实现标准ICloneable接口。如在本章开头提到的，这个类型定义了一个简单的方法Clone()：

```
public interface ICloneable
{
  object Clone();
}
```

很明显，不同对象的Clone()方法实现不一样。但基本功能差不多，都是将成员变量的值复制到同类型的新对象实例，然后向用户返回该实例。思考下面Point类的更新：

```
// Point现在支持克隆能力
public class Point : ICloneable
{
  public int X { get; set; }
  public int Y { get; set; }

  public Point(int xPos, int yPos) { X = xPos; Y = yPos; }
  public Point() { }

  //重写Object.ToString()
  public override string ToString()
  { return string.Format("X = {0}; Y = {1}", X, Y); }

  // 返回一个当前对象的副本
  public object Clone()
  { return new Point(this.X, this.Y); }
}
```

这样我们就可以创建完全独立的Point类型的副本了，如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Cloning *****\n");
  // 请注意Clone()返回了一个通用的对象类型
  // 读者需要显式转换来获取派生类型
  Point p3 = new Point(100, 100);
  Point p4 = (Point)p3.Clone();

  // 改变p4.x将不影响p3.x
  p4.X = 0;

  // 输出每个对象
  Console.WriteLine(p3);
  Console.WriteLine(p4);
  Console.ReadLine();
}
```

虽然现在的Point实现可以满足需要，但我们再做一些改进。因为Point类型并不包含内部引用类型变量，可以按如下代码所示简化Clone()方法的实现：

```
public object Clone()
{
  // 逐个复制每个Point字段成员
  return this.MemberwiseClone();
}
```

请注意，如果Point包含任何引用类型成员变量，MemberwiseClone()将这些引用复制到对象中（即浅复制）。如果读者想要支持真正的深复制，需要在克隆过程中创建任何引用类型变量的新实例，让我们来看看示例。

更复杂的克隆示例

假定Point类包含一个PointDescription类型的引用类型成员变量。这个类保存一个点的友好名称（friendly name）和表示为一个System.Guid的标识号。（对没有COM背景的读者，这里解释一下，全局唯一标识符[GUID]是一个静态的唯一的128位数字。）下面是具体实现：

```
// 这个类定义了一个点
public class PointDescription
{
  public string PetName {get; set;}
  public Guid PointID {get; set;}

  public PointDescription()
  {
    PetName = "No-name";
    PointID = Guid.NewGuid();
  }
}
```

之前对Point类的更新包括修改ToString()，以说明状态数据、定义并创建PointDescription引用类型。为了允许外部对象为Point创建一个昵称，要修改传进重载构造函数的参数：

```
public class Point : ICloneable
{
  public int X { get; set; }
  public int Y { get; set; }
  public PointDescription desc = new PointDescription();

  public Point(int xPos, int yPos, string petName)
  {
    X = xPos; Y = yPos;
    desc.PetName = petName;
  }
  public Point(int xPos, int yPos)
  {
    X = xPos; Y = yPos;
  }
  public Point() { }

  // 重写Object.ToString()
  public override string ToString()
  {
    return string.Format("X = {0}; Y = {1}; Name = {2};\nID = {3}\n",
    X, Y, desc.PetName, desc.PointID);
  }
```

8

```
  // 返回当前对象的副本
  public object Clone()
  { return this.MemberwiseClone(); }
}
```

注意，至今我们仍没有修改Clone()方法。所以，如果对象用户使用当前实现请求复制，将得到一个逐项浅复制。为便于说明，假定读者已经修改了Main()方法，具体如下：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Cloning *****\n");
  Console.WriteLine("Cloned p3 and stored new Point in p4");
  Point p3 = new Point(100, 100, "Jane");
  Point p4 = (Point)p3.Clone();

  Console.WriteLine("Before modification:");
  Console.WriteLine("p3: {0}", p3);
  Console.WriteLine("p4: {0}", p4);
  p4.desc.PetName = "My new Point";
  p4.X = 9;

  Console.WriteLine("\nChanged p4.desc.petName and p4.X");
  Console.WriteLine("After modification:");
  Console.WriteLine("p3: {0}", p3);
  Console.WriteLine("p4: {0}", p4);
  Console.ReadLine();
}
```

注意，当值类型被改变时，内部引用类型保存的是修改后的值，因为它们在内存中"指向"同样的对象（特别需要注意的是，这些对象的昵称现在是My new Point）。输出结果如下所示：

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509


Changed p4.desc.petName and p4.X
After modification:
p3: X = 100; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 9; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
```

为使Clone()方法得到内部引用类型的深复制，需要设定由MemberwiseClone()返回的对象来表示当前点的名称。System.Guid类型事实上是一个结构，数值数据被真正复制过来了。下面是一种可行的实现：

```
// 现在需要调整PointDescription成员
public object Clone()
```

```
{
  // 首先获取浅复制
  Point newPoint = (Point)this.MemberwiseClone();

  // 然后填充间距
  PointDescription currentDesc = new PointDescription();
  currentDesc.PetName = this.desc.PetName;
  newPoint.desc = currentDesc;
  return newPoint;
}
```

如果再运行一次这个程序并查看输出结果（如下所示），你将会看到返回自Clone()的Point确实复制了它的内部引用类型成员变量（注意现在p3和p4的昵称都是不同的）。

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 100; Y = 100; Name = Jane;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a


Changed p4.desc.petName and p4.X
After modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 9; Y = 100; Name = My new Point;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a
```

概括一下克隆过程。如果有一个仅包含值类型的类或结构，使用MemberwiseClone()实现Clone()方法。如果有一个保存其他引用类型的自定义类型，需要建立一个考虑了每个引用类型成员变量的新对象。

源代码 CloneablePoint项目的源代码位于Chapter 8子目录下。

8.13 构建可比较的对象（IComparable）

System.IComparable接口指定了一种允许一个对象可基于某些特定键值进行排序的行为。下面是正式定义：

```
// 这个接口允许一个对象指定它与类似对象的关系
public interface IComparable
{
  int CompareTo(object o);
}
```

说明 该接口的泛型版本（IComparable<T>）提供了类型更加安全的方式处理对象间的比较。你将在第9章中学习泛型的知识。

假定构建了一个名为ComparableCar的新控制台应用程序项目，它修改的Car类型如下所示（这里只是添加了一个新成员变量来表示每个Car的不同ID，以及获取和设置这个值的方法）：

```
public class Car
{
...
  public int CarID {get; set;}
  public Car(string name, int currSp, int id)
  {
    CurrentSpeed = currSp;
    PetName = name;
    CarID = id;
  }
...
}
```

假设你有一个如下所示的Car对象的数组：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Sorting *****\n");

  // 建立一个Car对象的数组
  Car[] myAutos = new Car[5];
  myAutos[0] = new Car("Rusty", 80, 1);
  myAutos[1] = new Car("Mary", 40, 234);
  myAutos[2] = new Car("Viper", 40, 34);
  myAutos[3] = new Car("Mel", 40, 4);
  myAutos[4] = new Car("Chucky", 40, 5);

  Console.ReadLine();
}
```

System.Array类定义了一个名为Sort()的静态方法。在内置类型（int、short、string等）上调用这个方法的时候，可以以数字/字母顺序对数组中的项排序，因为这些内置数据类型实现了IComparable。但我们按如下代码向Car对象数组发送一个Sort()方法，又会怎样呢？

```
// 给我的汽车排序吗
Array.Sort(myAutos);
```

运行这个测试，将发现一个运行时异常，因为Car类并不支持这个接口。构建自定义类型的时候，可以实现IComparable以使该类型数组可被排序。充实CompareTo()的细节时，需要决定排序操作的基准。对于Car类型来说，内部的CarID看上去是最合逻辑的选择：

```
// Car类的迭代可以基于CarID进行排序
public class Car : IComparable
{
...
  // IComparable的实现
  int IComparable.CompareTo(object obj)
  {
    Car temp = obj as Car;
    if (temp != null)
```

```
    {
      if (this.CarID > temp.CarID)
        return 1;
      if (this.CarID < temp.CarID)
        return -1;
      else
        return 0;
    }
    else
      throw new ArgumentException("Parameter is not a Car!");
  }
}
```

可以看到，CompareTo()方法背后的逻辑是，根据某个特定数据字段比较传入的对象与当前实例。CompareTo()方法的返回值被用来判断这个类型小于、大于或是等于它所比较的对象（如表8-1所示）。

表8-1 CompareTo()方法的返回值

CompareTo()方法的返回值	作　用
任何小于0的数字	这个实例在指定对象之前
0	这个实例等于指定对象
任何大于0的数字	这个实例在指定对象之后

由于C# int数据类型（只是CLR System.Int32的简写形式）实现了IComparable，我们就可以按如下所示的方法实现Car的CompareTo()方法：

```
int IComparable.CompareTo(object obj)
{
  Car temp = obj as Car;
  if (temp != null)
    return this.CarID.CompareTo(temp.CarID);
  else
    throw new ArgumentException("Parameter is not a Car!");
}
```

既然Car类型已知道如何将它自己和类似对象进行对比，你可以写如下所示的用户代码：

```
// 执行IComparable接口
static void Main(string[] args)
{
  // 建立一个Car对象数组
...
  // 显示当前数组
  Console.WriteLine("Here is the unordered set of cars:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);

  // 现在，使用IComparable为它们排序
  Array.Sort(myAutos);
  Console.WriteLine();

  // 显示排序后的数组
  Console.WriteLine("Here is the ordered set of cars:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);
  Console.ReadLine();
}
```

下面显示了前面Main()方法的运行结果：

```
***** Fun with Object Sorting *****

Here is the unordered set of cars:
1 Rusty
234 Mary
34 Viper
4 Mel
5 Chucky

Here is the ordered set of cars:
1 Rusty
4 Mel
5 Chucky
34 Viper
234 Mary
```

8.13.1　指定多个排序顺序

在上面这个版本的Car类型中，我们使用了ID号作为排序的基准。另一个设计也许会用昵称作为排序算法的基准。那如果要构建一个既可通过ID排序又可通过昵称排序的Car类型，该怎么办呢？如果读者对这种行为感兴趣，就需要与另一个标准接口IComparer打交道。它按如下代码所示定义在System.Collections命名空间里：

```
// 比较两个对象的通用方法
interface IComparer
{
  int Compare(object o1, object o2);
}
```

说明　该接口的泛型版本（IComparer<T>）提供了类型更加安全的方式来处理对象间的比较。你将在第9章学习泛型的知识。

与IComparable接口不同，IComparer接口不是在要排序的类型（即Car）中，而是在许多辅助类中实现的，其中每个排序各有一个依据（如昵称、ID号等）。现在，Car类型已经知道如何基于内部ID号进行对比，所以允许对象用户通过昵称排序Car对象数组，需要另一个实现IComparer接口的辅助类。代码如下（确保在代码文件中导入System.Collections命名空间）：

```
// 这个辅助类用来通过昵称排序Car类型的数组
public class PetNameComparer : IComparer
{
  // 测试每个对象的昵称
  int IComparer.Compare(object o1, object o2)
  {
    Car t1 = o1 as Car;
    Car t2 = o2 as Car;
    if(t1 != null && t2 != null)
```

```
        return String.Compare(t1.PetName, t2.PetName);
      else
        throw new ArgumentException("Parameter is not a Car!");
    }
  }
```

这个对象用户代码可以使用辅助类。System.Array有许多重载的Sort()方法，其中有一个用来在对象上实现IComparer接口。

```
static void Main(string[] args)
{
...
  // 按照昵称进行排序
  Array.Sort(myAutos, new PetNameComparer());

  // 转储排序后的数组
  Console.WriteLine("Ordering by pet name:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);
...
}
```

8.13.2 自定义属性、自定义排序类型

值得指出的是，在通过特定数据字段排序Car类型的时候，可以使用自定义的静态属性辅助对象用户。假定Car类添加了一个静态只读属性SortByPetName，它返回一个实现了IComparer接口的对象的实例（本例中为PetNameComparer）：

```
// 现在可以使用一个自定义静态属性来返回正确的IComparer接口
public class Car : IComparable
{
  ...
  // 返回SortByPetName比较的属性
  public static IComparer SortByPetName
  { get { return (IComparer)new PetNameComparer(); } }
}
```

对象用户代码现在可以使用强关联属性按照昵称排序，而不是只能使用独立的PetNameComparer类型：

```
// 简洁明了地按照昵称排序
Array.Sort(myAutos, Car.SortByPetName);
```

源代码 ComparableCar项目的源代码位于Chapter 8子目录下。

很高兴现在大家不仅了解了如何定义和实现接口类型，而且知道了接口的用处是多么大。是的，接口出现在每个主要的.NET命名空间里，你将在本书其他章节中继续使用各种标准接口。

8.14 小结

接口可以被定义为抽象成员的集合。因为接口不提供任何实现细节，通常把接口看做某个类型支

持的行为。如果两个或更多类实现相同的接口，我们就可以以相同方式对待两个类型（又叫基于接口的多态），即使类型定义在独立的类继承体系中。

C#提供了interface关键字来允许我们定义新的接口。我们已经看到，类型可以使用逗号分隔列表支持任意多个接口。此外，也可以构建从多个基接口派生的接口。

除了构建自定义接口之外，.NET类库还定义了许多标准（即框架支持的）接口。可以看到，我们完全可以构建自定义类型来实现这些预定义的接口以获得许多有用特性，如克隆、排序以及枚举。

Part 4

第四部分

高级 C# 编程结构

本部分内容

第9章

集合与泛型

使用.NET平台创建的任何应用程序，都需要解决在内存中维护和操作一组数据点的问题。这些数据点可以来自任何地方，包括关系型数据库、本地文本文件、XML文件、Web服务调用，也可能是用户提供的输入。

自从.NET平台第一次发布之后，程序员经常使用System.Collections命名空间来以更灵活的方式管理数据。然而，自从.NET 2.0发布之后，C#编程语言就增强了，开始支持泛型。基于此，在基础类库中引入了一个以集合为中心的新命名空间：System.Collections.Generic命名空间。

本章将介绍不同集合（泛型和非泛型）命名空间的概况和.NET基础类库中的类型。你会看到，泛型容器与它们的非泛型版本相比，在很多方面要优越得多，因为它们提供了更好的类型安全和性能优势。在我们看到如何创建和操纵框架中的泛型项之后，本章余下部分会研究如何构建自己的泛型类型。此时，你将学习约束的作用和C# where关键字，它允许构建类型安全的容器。

9.1 集合类的动机

毫无疑问，我们用来保存应用程序中数据的最简单的容器就是数组。在第4章我们看到，C#数组可以定义一组具有固定上限的相同类型的项（包括System.Object类型的数组，它实际上表示任何数据类型的一个数组）。同时，第4章还介绍了所有C#数组变量都从System.Array类获得了大量功能。我们来快速回顾一下这些知识，看看下面的Main()方法，它创建了一个文本数据的数组，并通过各种方式操纵其内容。

```
static void Main(string[] args)
{
  //创建一个字符串数据的数组
  string[] strArray = {"First", "Second", "Third" };

  // Show number of items in array using Length property.
  Console.WriteLine("This array has {0} items.", strArray.Length);
  Console.WriteLine();

  // 使用枚举器显示内容
  foreach (string s in strArray)
  {
    Console.WriteLine("Array Entry: {0}", s);
  }
  Console.WriteLine();

  // 颠倒数组并打印
```

```
    Array.Reverse(strArray);
    foreach (string s in strArray)
    {
      Console.WriteLine("Array Entry: {0}", s);
    }

    Console.ReadLine();
  }
```

尽管基本数组可以用来管理少量固定大小的数据，但很多时候你都会需要一种更灵活的数据结构，如容器可以动态伸缩，或只保存满足某个标准的对象（如只保存派生自某个特定基类或实现了某个接口的对象）。在使用简单数组时，始终要记得它们是“固定大小的”。如果创建了一个包含三个项的数组，就只能得到三个项，因此下面的代码将产生一个运行时异常（IndexOutOfRangeException）：

```
static void Main(string[] args)
{
  //创建一个字符串数据的数组
  string[] strArray = { "First", "Second", "Third" };

  //想在最后添加新项？将得到运行时错误
  strArray[3] = "new item?";
  ...
}
```

为了摆脱简单数组的这些限制，.NET基础类库发布了很多包含集合类的命名空间。与简单的C#数组不同，集合类本身的尺寸是动态的，你可以在运行时插入或移除项。此外，很多集合类还提供了更强的类型安全，并且进行了高度优化，可以以内存高效的方式处理所包含的数据。在读完本章后你会发现，集合类可划分为两大种类：

- 非泛型集合（主要位于System.Collections命名空间）
- 泛型集合（主要位于System.Collections.Generic命名空间）

非泛型集合通常设计为操作System.Object类型，因此是非常松散类型的容器（尽管如此，有些非泛型集合还是专门用来操作特定类型的数据，如string对象）。相反，泛型集合更加类型安全，因为必须在创建时指定“类型的类型”。你将会看到，任何泛型项中的标记叫做“类型参数”，用尖括号括起（如List<T>）。本章稍后会介绍泛型的细节（以及很多优点）。现在，我们先来学习System.Collections和System.Collections.Specialized命名空间中的一些主要的非泛型集合类型。

9.1.1 System.Collections命名空间

在.NET平台最初发布时，程序员常常使用System.Collections命名空间中的非泛型集合类。该命名空间提供了很多类来管理和组织大量的数据。表9-1列出了一些常用的集合类，以及它们所实现的核心接口。

说明 任何使用.NET 2.0或更高版本创建的项目都应该放弃使用System.Collections中的类，而使用System.Collections.Generic中的类。不过，有必要了解一些非泛型集合类的基础知识，因为你可能要使用它维护一些遗留软件。

表9-1　System.Collections中的常用类

System.Collections类	作　用	实现的关键接口
ArrayList	表示动态大小的对象集合，其中的对象是按顺序列出的	IList、ICollection、IEnumerable和ICloneable
BitArray	管理位值的简单数组，位值用布尔值表示，true表示该位打开（1），false表示该位关闭（0）	ICollection、IEnumerable和ICloneable
Hashtable	表示键值对的集合，按键的散列值进行组织	IDictionary、ICollection、IEnumerable和ICloneable
Queue	表示标准的先进先出（FIFO）队列	ICollection、ICloneable和IEnumerable
SortedList	表示键值对的集合，按键排序，可通过键和索引进行访问	IDictionary、ICollection、IEnumerable和ICloneable
Stack	后进先出（LIFO）的栈，提供压入和弹出功能	ICollection、ICloneable和IEnumerable

由这些基本集合类实现的接口可以洞悉整个功能。表9-2列出了这些关键接口的性质，其中一些已经在第8章中接触过。

表9-2　System.Collections中的类所支持的关键接口

System.Collections接口	作　用
ICollection	为所有非泛型集合类型定义基本特性（如大小、枚举、线程安全）
ICloneable	允许实现它的对象向调用者返回它的副本
IDictionary	允许非泛型集合对象使用名称/值对来表示其内容
IEnumerable	返回实现了IEnumerator接口（见下一行）的对象
IEnumerator	允许子类型以foreach形式迭代
IList	为顺序列表中的对象提供添加、移除和索引项的行为

示例演示：使用ArrayList

你可能以前使用过（或实现过）一些传统的数据结构，如栈、队列和列表。如果不是这样，我将在本章后面介绍泛型版本时再深入探讨它们的区别。在那之前，我们先来看看这个使用了ArrayList对象的Main()方法。注意，我们可以在运行时添加（或移除）项，容器将相应地自动改变大小：

```
//要访问ArrayList，必须引入System.Collections
static void Main(string[] args)
{
  ArrayList strArray = new ArrayList();
  strArray.AddRange(new string[] { "First", "Second", "Third" });

  //显示ArrayList中项个数目
  Console.WriteLine("This collection has {0} items.", strArray.Count);
  Console.WriteLine();
```

```
    //添加新项，并展示当前数目
    strArray.Add("Fourth!");
    Console.WriteLine("This collection has {0} items.", strArray.Count);

    //显示内容
    foreach (string s in strArray)
    {
      Console.WriteLine("Entry: {0}", s);
    }
    Console.WriteLine();
  }
```

正如你所猜测的那样，ArrayList类除了Count属性和AddRange()、Add()方法外，还包含很多有用的成员，详细内容请参考.NET Framework文档。此外，System.Collections下的其他类（Stack、Queue等）在.NET帮助系统中也包含完整的文档。

但必须指出的是，大多数.NET项目都不会使用System.Collections命名空间中的集合类！的确，这些年更常见的是使用System.Collections.Generic命名空间中的泛型类。鉴于此，我就不再赘述（或列举）System.Collections中的其他非泛型类了。

9.1.2 System.Collections.Specialized命名空间

System.Collections不是唯一包含非泛集合类的.NET命名空间。比如System.Collections.Specialized命名空间就定义了一些专有的集合类型。表9-3列出了这个特殊的以集合为核心的命名空间中一些十分有用的类型，它们都是非泛型的。

表9-3 System.Collections.Specialized中有用的类

System.Collections.Specialized中的类型	作　用
HybridDictionary	该类实现了IDictionary，在集合较小时使用ListDictionary，在集合变大后改用Hashtable
ListDictionary	在管理少量（10个左右）可随时改变的项时非常有用。它使用单链表来维护数据
StringCollection	该类提供了管理大型字符串集合的另一种方式
BitVector32	该类提供了一种简单的结构，用32位内存存储布尔值和小整数

除了这些核心类型，System.Collections.Specialized命名空间还包含了很多额外的接口和抽象基类，可以用来作为创建自定义集合类的起始点。这些“特定”的类型可能在某些情况下恰好是你的项目所需要的，这里就不逐一点评它们的用法了。此外，在许多情况下，你可能会发现System.Collections.Generic命名空间提供的类与上面这些功能相似，但还有其他一些好处。

说明 .NET基类库中还包含两个以集合为核心的命名空间（System.Collections.ObjectModel和System.Collections.Consurrent）。我们将在本章学习完泛型后介绍前一个命名空间。System.Collections.Concurrent提供了线程安全的集合类（参见第19章的多线程）。

9.2　非泛型集合的问题

尽管很多成功的.NET应用程序在多年前都使用这些“经典”的集合类（和接口）构建，但历史证明使用这些类型会导致相当多的问题。

第一个问题是，使用System.Collections和System.Collections.Specialized下的类会导致低性能的代码，特别是在操作数值数据时（如值类型）。稍后你将看到，当你在一个操作System.Object的非泛型集合类中存储结构时，CLR必须执行大量的内存转换操作，这会降低运行时的执行速度。

第二个问题是，这些经典的集合类不是类型安全的，因为它们是为了操作System.Object类而开发的，因此可以包含任何类型。如果.NET开发者需要创建高度类型安全的集合（如某容器只能容纳实现了某个接口的对象），唯一的选择是手工创建一个全新的集合类。这么做的工作量并不大，但却多少有点繁琐。

在学习如何在程序中使用泛型之前，研究一下非泛型集合类的问题会非常有用。这将有助于你更好地理解泛型要解决的首要问题。创建一个名为IssuesWithNongenericCollections的Console Application，然后在C#代码文件的顶端引入System.Collections命名空间：

```
using System.Collections;
```

9.2.1　性能问题

第4章介绍过，.NET平台支持两大类数据类型：值类型和引用类型。由于.NET定义了这两大类型，我们有时需要用一个类别的变量来表示另一个类别的变量。为此，C#提供了称为装箱的简单机制来实现，它可以把值类型数据保存在引用类型变量中。假设我们在SimpleBoxUnboxOperation()方法中创建了一个int类型的本地变量。在应用程序的过程中，如果要将值类型表示为引用类型，需要对值进行装箱，如下所示：

```
static void SimpleBoxUnboxOperation()
{
  //创建一个ValueType(int)变量
  int myInt = 25;

  //将int装箱为object引用
  object boxedInt = myInt;
}
```

装箱可以正式定义为：显式地将值类型分配给System.Object变量的过程。当我们对一个值进行装箱时，CLR就会在堆上分配新的对象并且将值类型的值（这里是25）复制到那个实例上。因此，返回给我们的就是新分配在堆上的对象的引用。

相反的操作可以通过拆箱来实现。拆箱就是把保存在对象引用中的值转换回栈上的相应值类型。从语法上来说，拆箱操作更像是转换操作。但它们在语义上却迥然不同。CLR会验证收到的值类型是不是等价于装箱的类型，如果是，就将值复制回本地栈变量上。例如，如下拆箱操作会成功，因为boxedInt的实际类型确实是int：

```
static void SimpleBoxUnboxOperation()
{
```

```
  //创建一个ValueType(int)变量
  int myInt = 25;

  //将int装箱为object引用
  object boxedInt = myInt;

  //将引用拆箱为相应的int
  int unboxedInt = (int)boxedInt;
}
```

当C#编译器发现装箱/拆箱语法时，所生成的CIL代码包含box/unbox操作码。用ildasm.exe查看编译后的程序集，结果如下：

```
.method private hidebysig static void SimpleBoxUnboxOperation() cil managed
{
  // 代码大小19 (0x13)
  .maxstack  1
  .locals init ([0] int32 myInt, [1] object boxedInt, [2] int32 unboxedInt)
  IL_0000:  nop
  IL_0001:  ldc.i4.s 25
  IL_0003:  stloc.0
  IL_0004:  ldloc.0
  IL_0005:  box  [mscorlib]System.Int32
  IL_000a:  stloc.1
  IL_000b:  ldloc.1
  IL_000c:  unbox.any [mscorlib]System.Int32
  IL_0011:  stloc.2
  IL_0012:  ret
} // Program::SimpleBoxUnboxOperation方法结束
```

记住，与执行通常的转换不同，拆箱必须回到合适的数据类型。如果尝试将数据拆箱为不正确的变量，将抛出InvalidCastException异常。为了确保安全，你需要将每个拆箱操作都包裹在try/catch逻辑中。但这样的工作量又太大了。由于尝试将装箱后的int拆箱为long，下面的代码将抛出一个错误：

```
static void SimpleBoxUnboxOperation()
{
  // 创建一个ValueType(int)变量
  int myInt = 25;

  // 将int装箱为object引用
  object boxedInt = myInt;

  // 拆箱为错误的数据类型将触发运行时异常
  try
  {
    long unboxedInt = (long)boxedInt;
  }
  catch (InvalidCastException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

乍看上去，装箱和拆箱看似是一个普通的语言特性，其学术意义大于实际意义。毕竟，你不会像示例中那样将本地值类型保存为本地object变量。但其实装（拆）箱过程很有用，因为我们可以将所有东西都当成System.Object，而CLR会自己负责内存相关的细节。

9

为了看看这项技术的实际用途，假设我们创建了非泛型的System.Collections.ArrayList来保存数值（分配在栈上的）数据。如果研究ArrayList成员，就会发现它们所操作的是System.Object数据。来看看Add()、Insert()、Remove()方法以及类索引器：

```
public class ArrayList : object,
  IList, ICollection, IEnumerable, ICloneable
{
...
  public virtual int Add(object value);
  public virtual void Insert(int index, object value);
  public virtual void Remove(object obj);
  public virtual object this[int index] {get; set; }
}
```

ArrayList所操作的是object，这表示数据分配在堆上，因此下面的代码可能看上去有些奇怪，因为它不但可以通过编译，而且执行时也不会抛出错误：

```
static void WorkWithArrayList()
{
  // 在传递给需要object的方法时，值类型会自动装箱
  ArrayList myInts = new ArrayList();
  myInts.Add(10);
  myInts.Add(20);
  myInts.Add(35);
}
```

尽管你把数字数据直接传入要求object的方法，但运行时会自动将栈数据进行装箱。然后，如果希望使用类型索引器从ArrayList中获取项，则必须使用转换操作，将堆分配的对象拆箱成栈分配的整数。记住，ArrayList的索引器返回System.Object，不是System.Int32：

```
static void WorkWithArrayList()
{
  // 在传递给要求object的成员时，值类型将自动装箱
  ArrayList myInts = new ArrayList();
  myInts.Add(10);
  myInts.Add(20);
  myInts.Add(35);

  // 当将object转换回栈数据时，会发生拆箱
  int i = (int)myInts[0];

  // 由于WriteLine()要求object类型，因此再次发生装箱操作
  Console.WriteLine("Value of your int: {0}",  i);
}
```

再次注意，在调用ArrayList.Add()之前，栈分配的System.Int32被装箱，以便它被传递给所需的System.Object。同样，通过类型索引器从ArrayList获取数据时，System.Object将被拆箱为System.Int32，只有当传递给Console.WriteLine()方法时，才会被再次装箱，这是因为该方法操作的也是System.Object变量。

尽管装箱和拆箱对程序员来说很方便，但是这种方式带来的堆/栈内存转移会导致性能问题（执行速度和代码多少），并且也缺乏类型安全。为了理解性能问题，思考一下在装箱和拆箱一个整数时发生的步骤，具体如下所示。

(1) 必须在托管堆上分配一个新对象。

(2) 基于栈数据的值必须被转移到新分配的内存位置。

(3) 在拆箱时，保存在堆对象中的值必须转移回栈。

(4) 堆上无用的对象（最后）会被回收。

尽管这个WorkWithArrayList()方法不会导致明显的性能瓶颈，但如果程序维护的ArrayList包含了几千个整数就很是问题了。理想情况下，我们应该可以在没有任何性能问题的容器中操作栈数据，并且在获取数据时也不必使用try/catch作用域（这正是泛型所实现的）。

9.2.2 类型安全问题

在介绍拆箱操作时，我们提到了类型安全问题。我们必须将数据拆箱为装箱之前声明的类型。但是，你还必须记住非泛型世界中的另一个类型安全问题：由于System.Collections中的大多数类所操作的都是System.Object，因此它们可以容纳任何类型。例如，下面方法中的ArrayList包含一些不相关的数据：

```
static void ArrayListOfRandomObjects()
{
  // ArrayList可以保存任何类型
  ArrayList allMyObjects = new ArrayList();
  allMyObjects.Add(true);
  allMyObjects.Add(new OperatingSystem(PlatformID.MacOSX, new Version(10, 0)));
  allMyObjects.Add(66);
  allMyObjects.Add(3.14);
}
```

某些情况下，你可能需要极其灵活的容器来容纳任何东西（如本例）。但大多数情况下，你还是需要一个类型安全的容器来操作特定的数据类型。例如，你可能需要只能容纳数据库连接、位图或IPointy兼容对象的容器。

在泛型之前，要解决类型安全问题的唯一方法是手工创建自定义（强类型的）集合类。假设你要创建一个只包含Person类型对象的自定义集合：

```
public class Person
{
  public int Age {get; set;}
  public string FirstName {get; set;}
  public string LastName {get; set;}

  public Person(){}
  public Person(string firstName, string lastName, int age)
  {
    Age = age;
    FirstName = firstName;
    LastName = lastName;
  }

  public override string ToString()
  {
    return string.Format("Name: {0} {1}, Age: {2}",
      FirstName, LastName, Age);
  }
}
```

要构建只容纳Person对象的集合，我们可以在PersonCollection类中定义一个System.Collections.

ArrayList成员变量，并且配置所有成员操作强类型的Person对象而不是System.Object类型。下面是一个简单的示例（一个产品级的自定义集合可支持更多额外的成员，并可扩展System.Collections或者System.Collections.Specialized命名空间中的某个抽象基类）：

```
public class PersonCollection : IEnumerable
{
  private ArrayList arPeople = new ArrayList();

  // 为调用者进行转换
  public Person GetPerson(int pos)
  { return (Person)arPeople[pos]; }

  // 只插入Person对象
  public void AddPerson(Person p)
  { arPeople.Add(p); }

  public void ClearPeople()
  { arPeople.Clear(); }

  public int Count
  { get { return arPeople.Count; } }

  // 支持foreach枚举
  IEnumerator IEnumerable.GetEnumerator()
  { return arPeople.GetEnumerator(); }
}
```

注意，为了允许对包含的项进行foreach迭代，PersonCollection类实现了IEnumerable接口。还要注意，GetPerson()和AddPerson()方法已经被定义为只操作Person对象（不是位图、字符串、数据库连接等）。定义了这些类型以后，就不用担心类型安全了，因为C#编译器会检查任何尝试插入不兼容数据类型的请求。

```
static void UsePersonCollection()
{
  Console.WriteLine("***** Custom Person Collection *****\n");
  PersonCollection myPeople = new PersonCollection();
  myPeople.AddPerson(new Person("Homer", "Simpson", 40));
  myPeople.AddPerson(new Person("Marge", "Simpson", 38));
  myPeople.AddPerson(new Person("Lisa", "Simpson", 9));
  myPeople.AddPerson(new Person("Bart", "Simpson", 7));
  myPeople.AddPerson(new Person("Maggie", "Simpson", 2));

  // 这会产生编译时错误
  // myPeople.AddPerson(new Car());

  foreach (Person p in myPeople)
    Console.WriteLine(p);
}
```

虽然自定义的集合可以确保类型安全，但是如果使用这个方法，就必须为每一个希望包含的类型创建一个（基本一样）自定义集合。因此，如果我们需要一个自定义集合来操作从Car基类派生的任何类，就需要构建非常相似的集合类：

```
public class CarCollection : IEnumerable
{
  private ArrayList arCars = new ArrayList();
```

```
    // 为调用者进行转换
    public Car GetCar(int pos)
    { return (Car) arCars[pos]; }

    // 只插入Car对象
    public void AddCar(Car c)
    { arCars.Add(c); }

    public void ClearCars()
    { arCars.Clear(); }

    public int Count
    { get { return arCars.Count; } }

    // 支持foreach枚举
    IEnumerator IEnumerable.GetEnumerator()
    { return arCars.GetEnumerator(); }
  }
```

然而，这些自定义集合类并没有消除装箱/拆箱的损失。即使我们创建叫做IntCollection的自定义集合，只处理System.Int32数据项，我们还是要分配某个类型的对象来保存数据（如System.Array和ArrayList等）：

```
  public class IntCollection : IEnumerable
  {
    private ArrayList arInts = new ArrayList();

    // 为调用者拆箱
    public int GetInt(int pos)
    { return (int)arInts[pos]; }

    // 装箱操作
    public void AddInt(int i)
    { arInts.Add(i); }

    public void ClearInts()
    { arInts.Clear(); }

    public int Count
    { get { return arInts.Count; } }

    IEnumerator IEnumerable.GetEnumerator()
    { return arInts.GetEnumerator(); }
  }
```

不管选择哪个类型来保存整数，我们都不能避免使用非泛型容器带来的装箱问题。

9.2.3 初识泛型集合

使用泛型集合类可以解决上面所有的问题，包括装箱/拆箱损耗和类型安全缺失。另外，基本上不需要手工构建自定义（泛型）集合类。与其构建包含人、汽车和整数的类，不如使用泛型集合类并指定类型。

考虑如下的方法，它使用泛型List<T>类（位于System.Collections.Generic命名空间）以强类型的方式保存不同的类型（不用在意泛型语法的细节）：

```
static void UseGenericList()
{
  Console.WriteLine("***** Fun with Generics *****\n");

  // 该List<>只能容纳Person对象
  List<Person> morePeople = new List<Person>();
  morePeople.Add(new Person ("Frank", "Black", 50));
  Console.WriteLine(morePeople[0]);

  // 该List<>只能容纳整数
  List<int> moreInts = new List<int>();
  moreInts.Add(10);
  moreInts.Add(2);
  int sum = moreInts[0] + moreInts[1];

  // 编译时错误！不能将Person对象添加到整型列表中
  // moreInts.Add(new Person());
}
```

第一个List<T>对象只能包含Person对象。因此，在从容器中获取项时不必进行转换，这使得这种方法更加类型安全。第二个List<T>只能包含整型，所有的整数都分配在栈上。也就是说，非泛型的ArrayList中没有那种隐藏的装箱或拆箱。与非泛型容器相比，泛型容器的一些优势如下所示。

- 泛型提供了更好的性能，因为它们不会导致装箱或拆箱的损耗。
- 泛型更类型安全，因为它们只包含我们指定的类型。
- 泛型大幅减少了构建自定义集合类型的需要，因为当创建泛型容器时指定了“类型的类型”。

源代码 IssuesWithNonGenericCollections项目的源代码位于Chapter 9子目录下。

9.3 泛型类型参数的作用

在.NET基础类库的每个命名空间中几乎都可以看到泛型类、接口、结构和委托。大家也非常清楚，比起单纯定义集合类，泛型的用处要大得多。事实确实是这样，你会看到本书剩余部分会以各种理由用到泛型。

说明 只有类、结构、接口和委托可以使用泛型，枚举类型不可以。

当你在.NET Framework 4.5 SDK文档或Visual Studio对象浏览器中看到泛型项时，你会发现其表现形式为一对尖括号包着内部的字母或其他标记。图9-1显示了System.Collections.Generic命名空间中的一些泛型项，包括突出显示的List<T>类。

尖括号中标记的正式名称为类型参数，但你也可以通俗地将其称为占位符。<T>符号读作*of T*。因此IEnumerable<T>读作*IEnumerable of T*，或者也可以称其为类型*T*的枚举。

说明 类型参数（占位符）的名称可以随意取，它取决于创建该泛型项的人。但通常情况下，*T*表示类型，*TKey*或*K*表示键，*TValue*或*V*表示值。

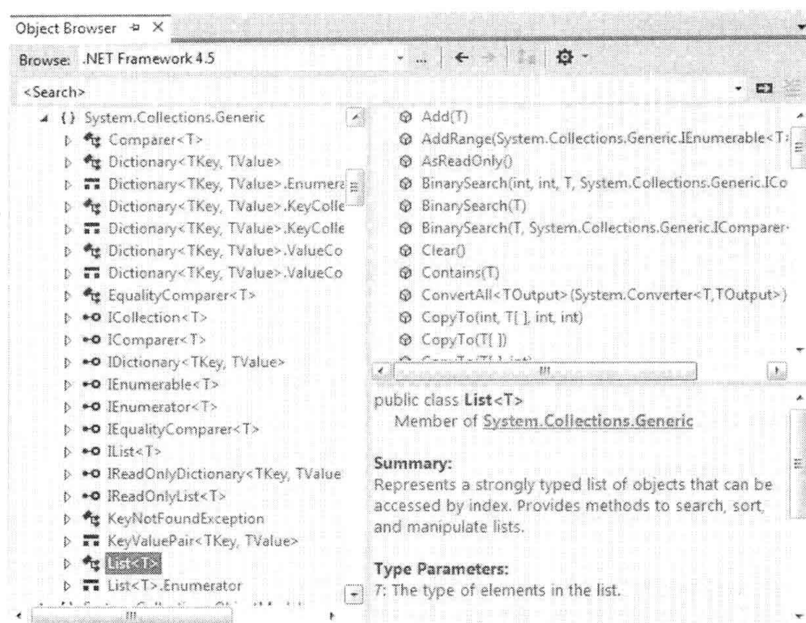


图9-1　支持类型参数的泛型项

在创建泛型对象、实现泛型接口或调用泛型成员时，为泛型参数提供的值是由开发者决定的。在本章中你会看到大量的示例。但为了打好基础，我们先来看看如何与泛型类型和泛型成员进行交互。

9.3.1　为泛型类/结构指定类型参数

在创建泛型类或泛型结构的实例时，需要在声明变量和调用构造函数时指定类型参数。在之前的代码示例中，UseGenericList()定义了两个List<T>对象：

```
// 该List<>只能容纳Person对象
List<Person> morePeople = new List<Person>();
```

这个代码片段可以读成*a List<> of T，T*为Person类型。或者更简单地读作Person对象列表。为泛型项指定了类型参数后，就不能更改了（记住，泛型是类型安全的）。在为泛型类或结构指定类型参数时，所有的占位符都将替换为你提供的值。

如果用Visual Studio对象浏览器查看泛型List<T>类的完整声明，你会发现占位符T遍布整个List<T>类型的定义。以下是部分代码（注意粗体的部分）：

```
// List<T>类的部分代码
namespace System.Collections.Generic
{
  public class List<T> :
    IList<T>, ICollection<T>, IEnumerable<T>,IReadOnlyList<T>
    IList, ICollection, IEnumerable
  {
...
```

```
    public void Add(T item);
    public ReadOnlyCollection<T> AsReadOnly();
    public int BinarySearch(T item);
    public bool Contains(T item);
    public void CopyTo(T[] array);
    public int FindIndex(System.Predicate<T> match);
    public T FindLast(System.Predicate<T> match);
    public bool Remove(T item);
    public int RemoveAll(System.Predicate<T> match);
    public T[] ToArray();
    public bool TrueForAll(System.Predicate<T> match);
    public T this[int index] { get; set; }
  }
}
```

当创建指定Person对象的List<T>时，就如同List<T>类型的定义如下：

```
namespace System.Collections.Generic
{
  public class List<Person> :
    IList<Person>, ICollection<Person>, IEnumerable<Person>,IReadOnlyList<Person>
    IList, ICollection, IEnumerable
  {
...
    public void Add(Person item);
    public ReadOnlyCollection<Person> AsReadOnly();
    public int BinarySearch(Person item);
    public bool Contains(Person item);
    public void CopyTo(Person[] array);
    public int FindIndex(System.Predicate<Person> match);
    public Person FindLast(System.Predicate<Person> match);
    public bool Remove(Person item);
    public int RemoveAll(System.Predicate<Person> match);
    public Person[] ToArray();
    public bool TrueForAll(System.Predicate<Person> match);
    public Person this[int index] { get; set; }
  }
}
```

当然，在创建泛型List<T>变量时，编译器并没有为List<T>类创建全新的实现，而是只处理你实际调用的泛型类型的成员。

9.3.2 为泛型成员指定类型参数

非泛型类或结构都支持泛型成员（如方法和属性）。因此在调用这种方法时，你仍然需要指定占位符的值。例如，System.Array支持一些泛型方法。如现在的非泛型Sort()静态方法包含一个相应的泛型方法Sort<T>()。考虑下面的代码段，其中T为int类型：

```
int[] myInts = { 10, 4, 2, 33, 93 };

// 为Sort<>()泛型方法指定占位符
Array.Sort<int>(myInts);

foreach (int i  in myInts)
{
  Console.WriteLine(i);
}
```

9.3.3 为泛型接口指定类型参数

在构建支持不同框架行为（如克隆、排序和枚举）的类或结构时，实现泛型接口是很常见的。我们在第8章中学习了一些非泛型接口，如IComparable、IEnumerable、IEnumerator和IComparer。非泛型的IComparable接口的定义如下：

```
public interface IComparable
{
  int CompareTo(object obj);
}
```

在第8章中，我们在Car类上实现了该接口，使之能够在标准数组中排序。但是，代码需要进行很多运行时检查和转换操作，因为参数是通用的System.Object：

```
public class Car : IComparable
{
...
  // IComparable的实现
  int IComparable.CompareTo(object obj)
  {
    Car temp = obj as Car;
    if (temp != null)
    {
      if (this.CarID > temp.CarID)
        return 1;
      if (this.CarID < temp.CarID)
        return -1;
      else
        return 0;
    }
    else
      throw new ArgumentException("Parameter is not a Car!");
  }
}
```

现在，假设我们使用与该接口对应的泛型：

```
public interface IComparable<T>
{
  int CompareTo(T obj);
}
```

这时，实现的代码会变得十分整洁：

```
public class Car : IComparable<Car>
{
...
  // IComparable<T>的实现
  int IComparable<Car>.CompareTo(Car obj)
  {
    if (this.CarID > obj.CarID)
      return 1;
    if (this.CarID < obj.CarID)
      return -1;
    else
      return 0;
  }
}
```

这里你不用判断传入的参数是否为Car，因为它只能为Car。如果传入的数据类型不兼容，将得到编译时错误。现在我们已经了解了与泛型项交互的方式和类型参数（占位符）的作用，下面就来学习System.Collections.Generic命名空间下的类和接口。

9.4　System.Collections.Generic 命名空间

我们构建.NET应用程序时需要一种管理内存数据的方式，System.Collections.Generic就是专门做这个的。在本章开始，我简单提到了一些非泛型接口，它们是由一些非泛型集合类实现的。不要奇怪，System.Collections.Generic命名空间中为大多数非泛型接口都定义了泛型版本。

实际上，你会发现很多泛型接口都扩展了它们的非泛型版本！这看上去很怪异。不过，这么做可以使那些实现类支持非泛型版本中的旧功能。例如，IEnumerable<T>扩展了IEnumerable。表9-4列出了在使用泛型集合类时经常遇到的核心泛型接口。

表9-4　System.Collections.Generic中的类支持的核心接口

System.Collections.Generic接口	作　用
ICollection<T>	为所有泛型集合类型定义基本特性（如大小、枚举、线程安全）
IComparer<T>	定义比较对象的方式
IDictionary<TKey, TValue>	允许泛型集合对象使用名称/值对来表示其内容
IEnumerable<T>	为给定对象返回IEnumerator<T>接口
IEnumerator<T>	允许泛型集合以foreach形式迭代
IList<T>	为顺序列表中的对象提供添加、移除和索引项的行为
ISet<T>	为抽象的集提供基接口

System.Collections.Generic命名空间还定义了一些实现这些核心接口的类。表9-5描述了该命名空间中的常用类、它们实现的接口和基本功能。

表9-5　System.Collections.Generic中的类

泛　型　类	支持的关键接口	作　用
Dictionary<TKey,TValue>	ICollection<T>、IDictionary<TKey, TValue>、IEnumerable<T>	一个名/值对泛型集合
LinkedList<T>	ICollection<T>、IEnumerable<T>	一个双向链表的泛型实现
List<T>	ICollection<T>、IEnumerable<T>、IList<T>	一个可动态改变大小的顺序列表
Queue<T>	ICollection（不是笔误，就是非泛型集合接口）、IEnumerable<T>	一个先入先出（FIFO）列表的泛型实现
SortedDictionary<TKey,TValue>	ICollection<T>、IDictionary<TKey, TValue>、IEnumerable<T>	一个排序的名称/值对集合的泛型实现
SortedSet<T>	ICollection<T>、IEnumerable<T>、ISet<T>	一个排序的不重复的对象集合
Stack<T>	ICollection（不是笔误，就是非泛型集合接口）、IEnumerable<T>	一个后入先出（LIFO）列表的泛型实现

System.Collections.Generic命名空间还定义了许多与特定容器联合使用的辅助类与结构。举例来说，LinkedListNode<T>类型表示泛型LinkedList<T>中容纳的节点。如果试图使用不存在的键从容器中取出一项，将触发KeyNotFoundException异常。

还要指出一点，System.Collections.Generic命名空间不仅仅在mscorlib.dll和System.dll程序集中添加了新的类型。例如，在System.Core.dll中添加了HashSet<T>类。有关System.Collections.Generic命名空间更详细的内容，请参考.NET Framework文档。

我们下一步的任务是学习如何使用这些泛型集合类。不过在这之前，我要介绍一个（.NET 3.5中引入的）C#语言特性，它简化了填充泛型（和非泛型）数据集合容器的方式。

9.4.1 集合初始化语法

在第4章中，我们学习了*对象初始化语法*，它允许我们在构造变量时设置其属性。与此密切相关的是*集合初始化语法*。这个C#语言特性让你可以用与填充基础数组类似的语法，来填充ArrayList或List<T>等容器。

说明 只能对支持Add()方法的类使用集合初始化语法，这是ICollection<T>/ICollection接口决定的。

考虑如下的示例：

```
// 初始化标准的数组
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// 初始化整数的泛型List<>
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// 使用数字数据初始化ArrayList
ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

如果容器操作的是类或结构的集合，你可以将对象初始化语法与集合初始化语法混合使用，以产生一些功能代码。你也许还记得第5章中的Point类，它定义了两个属性X和Y。如果要构建Point对象的泛型List<T>，可以编写如下的代码：

```
List<Point> myListOfPoints = new List<Point>
{
  new Point { X = 2, Y = 2 },
  new Point { X = 3, Y = 3 },
  new Point(PointColor.BloodRed){ X = 4, Y = 4 }
};

foreach (var pt in myListOfPoints)
{
  Console.WriteLine(pt);
}
```

这种语法的好处也是减少键盘输入。如果不注意格式的话，这种嵌套的大括号会降低可读性。但是，如果没有集合初始化语法，填充下面这个Rectangle的List<T>需要的代码量将是非常惊人的（你也许还记得第4章中我们创建的Rectangle类，它包含的两个属性都封装了Point对象）：

9

```
List<Rectangle> myListOfRects = new List<Rectangle>

{
  new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                BottomRight = new Point { X = 200, Y = 200}},
  new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                BottomRight = new Point { X = 100, Y = 100}},
  new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                BottomRight = new Point { X = 90, Y = 75}}
};

foreach (var r in myListOfRects)
{
  Console.WriteLine(r);
}
```

9.4.2　使用List<T>类

创建一个全新的Console Application项目FunWithGenericCollections。注意初始的C#代码文件还引入了System.Collections.Generic命名空间。

我们要研究的第一个泛型类是List<T>，它在本章中已经出现一两次了。List<T>类是System.Collections.Generic命名空间中最常用的类型，因为它可以动态调整内容。为了演示该类型的基础，考虑下面这个Program类中的方法，该类使用List<T>操作本章前面提到的Person对象。这些Person对象定义了3个属性（Age、FirstName和LastName）和一个自定义ToString()实现：

```
static void UseGenericList()
{
  // 使用集合/对象初始化语法，构建一个Person对象的列表
  List<Person> people = new List<Person>()
  {
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
  };

  // 打印列表中项的个数
  Console.WriteLine("Items in list: {0}", people.Count);

  // 枚举列表
  foreach (Person p in people)
    Console.WriteLine(p);

  // 插入一个新Person
  Console.WriteLine("\n->Inserting new person.");
  people.Insert(2, new Person { FirstName = "Maggie", LastName = "Simpson", Age = 2 });
  Console.WriteLine("Items in list: {0}", people.Count);

  // 将数据复制到新的数组中
  Person[] arrayOfPeople = people.ToArray();
  for (int i = 0; i < arrayOfPeople.Length; i++)
  {
    Console.WriteLine("First Names: {0}", arrayOfPeople[i].FirstName);
  }
}
```

这里我们使用初始化语法将对象填入List<T>，将其作为调用*n*次Add()方法的简写形式。打印集合的项数和枚举各个项之后，调用了Insert()方法。如你所见，Insert()可以向List<T>中的指定索引位置插入一个新项。

最后调用了ToArray()方法，它基于原始的List<T>，返回一个Person对象数组。我们使用数组索引器语法，对该数组再次执行循环。在Main()中调用该方法，将得到如下输出结果：

```
***** Fun with Generic Collections *****

Items in list: 4
Name: Homer Simpson, Age: 47
Name: Marge Simpson, Age: 45
Name: Lisa Simpson, Age: 9
Name: Bart Simpson, Age: 8

->Inserting new person.
Items in list: 5
First Names: Homer
First Names: Marge
First Names: Maggie
First Names: Lisa
First Names: Bart
```

List<T>类还定义了其他一些有趣的成员，更多信息请参考.NET Framework文档。接下来，我们来学习更多的泛型集合，如Stack<T>、Queue<T>和SortedSet<T>。这将有助于你理解如何选择保存用户自定义数据的方式。

9.4.3　使用Stack<T>类

Stack<T>类表示以后进先出的方式维护数据的集合。它包含Push()和Pop()方法，可以向栈压入数据或从栈移除数据。下面的代码创建了一个Person对象的栈：

```
static void UseGenericStack()
{
  Stack<Person> stackOfPeople = new Stack<Person>();
  stackOfPeople.Push(new Person
    { FirstName = "Homer", LastName = "Simpson", Age = 47 });
  stackOfPeople.Push(new Person
    { FirstName = "Marge", LastName = "Simpson", Age = 45 });
  stackOfPeople.Push(new Person
    { FirstName = "Lisa", LastName = "Simpson", Age = 9 });

  // 观察栈顶的项，取出，再次观察
  Console.WriteLine("First person is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

  Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

  Console.WriteLine("\nFirst person item is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

  try
```

```
  {
    Console.WriteLine("\nnFirst person is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine("\nError! {0}", ex.Message);
  }
}
```

这里，我们构建了包含3个Persen的栈，按名字的顺序添加：Homer、Marge和Lisa。在观察栈时，得到的永远是栈顶对象。因此，我们先调用Peek()来显示第三个Person对象。在一系列的Pop()和Peek()调用之后，栈最终为空，这时再调用Peek()和Pop()将触发系统异常。输出结果如下：

```
***** Fun with Generic Collections *****

First person is: Name: Lisa Simpson, Age: 9
Popped off Name: Lisa Simpson, Age: 9

First person is: Name: Marge Simpson, Age: 45
Popped off Name: Marge Simpson, Age: 45

First person item is: Name: Homer Simpson, Age: 47
Popped off Name: Homer Simpson, Age: 47

Error! Stack empty.
```

9.4.4　使用Queue<T>类

队列是确保以先进先出的方式访问数据的容器。悲剧的是，我们人类整天都在队列：在银行排队，在电影院排队，在咖啡馆排队。如果你需要对一个以先到先得方式处理数据的场景进行建模，Queue<T>类是很适合的。除了接口提供的功能以外，Queue还定义了如表9-6所示的关键成员。

表9-6　Queue<T>类型的成员

Queue<T>中的选择成员	作　用
Dequeue()	移除并返回Queue<T>开始处的对象
Enqueue()	在Queue<T>的末尾处添加一个对象
Peek()	返回Queue<T>开始处的对象，但不移除

现在来使用这些方法。我们再次使用Person类构建一个Queue<T>，来模仿一群正在排队点咖啡的人。首先，编写如下的静态辅助方法：

```
static void GetCoffee(Person p)
{
  Console.WriteLine("{0} got coffee!", p.FirstName);
}
```

现在假设你有另一个辅助方法，在其内部调用GetCoffee()：

```
static void UseGenericQueue()
{
  // 创建一个包含3个人的队列
  Queue<Person> peopleQ = new Queue<Person>();
  peopleQ.Enqueue(new Person {FirstName= "Homer",
    LastName="Simpson", Age=47});
  peopleQ.Enqueue(new Person {FirstName= "Marge",
    LastName="Simpson", Age=45});
  peopleQ.Enqueue(new Person {FirstName= "Lisa",
    LastName="Simpson", Age=9});

  // 观察队列中的第一个人
  Console.WriteLine("{0} is first in line!", peopleQ.Peek().FirstName);

  // 移除队列中的人
  GetCoffee(peopleQ.Dequeue());
  GetCoffee(peopleQ.Dequeue());
  GetCoffee(peopleQ.Dequeue());
  // 再次从队列中获取数据
  try
  {
    GetCoffee(peopleQ.Dequeue());
  }
  catch(InvalidOperationException e)
  {
    Console.WriteLine("Error! {0}", e.Message);
  }
}
```

我们在这里使用Enqueue()方法向Queue<T>类中插入了3个项。调用Peek()方法可以观察（但不移除）当前Queue中的第一项。最后，调用Dequeue()方法移除队列中的项，并将其发送给GetCoffee()辅助方法进行处理。注意，如果你对一个空队列进行移除操作，将抛出运行时异常。以下是调用该方法的输出结果：

```
***** Fun with Generic Collections *****

Homer is first in line!
Homer got coffee!
Marge got coffee!
Lisa got coffee!
Error! Queue empty.
```

9.4.5 使用SortedSet<T>类

我们要介绍的最后一个泛型集合类是.NET 4.0刚刚引入的SortedSet<T>类。SortedSet<T>类中的项是排序的，在插入和移除项之后，也能自动确保排序正确，因此该类十分有用。不过，你必须通知SortedSet<T>按何种方式进行排序，你可以向其构造函数传递一个实现了IComparer<T>泛型接口的参数。

创建一个全新的类SortPeopleByAge，实现IComparer<T>，其中T为Person类型。记住该接口定义了Compare()方法，可以包含任何需要的比较逻辑。下面是该类的简单实现：

```
class SortPeopleByAge : IComparer<Person>
{
  public int Compare(Person firstPerson, Person secondPerson)
```

```
  {
    if (firstPerson.Age > secondPerson.Age)
      return 1;
    if (firstPerson.Age < secondPerson.Age)
      return -1;
    else
      return 0;
  }
}
```

现在在Program类中添加下面的新方法，我们假设在Main()中调用该方法：

```
static void UseSortedSet()
{
  // 添加一些不同年龄的人
  SortedSet<Person> setOfPeople = new SortedSet<Person>(new SortPeopleByAge())
  {
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
  };

  // 各项是按照年龄排序的
  foreach (Person p in setOfPeople)
  {
    Console.WriteLine(p);
  }
  Console.WriteLine();

  // 添加一些具有不同年龄的人
  setOfPeople.Add(new Person { FirstName = "Saku", LastName = "Jones", Age = 1 });
  setOfPeople.Add(new Person { FirstName = "Mikko", LastName = "Jones", Age = 32 });

  // 仍然按照年龄排序
  foreach (Person p in setOfPeople)
  {
    Console.WriteLine(p);
  }
}
```

运行应用程序，列表中的对象将永远按Age属性的值排序，而与插入和移除对象的顺序无关：

```
***** Fun with Generic Collections *****

Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47

Name: Saku Jones, Age: 1
Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Mikko Jones, Age: 32
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47
```

源代码　FunWithGenericCollections项目的源代码位于Chapter 9子目录下。

9.5 System.Collections.ObjectModel 命名空间

现在你已经理解了如何使用泛型，下面我们来看一另一个以泛型为核心的命名空间System.Collections.ObjectModel。它是一个非常小的命名空间，只包含少量的类。表9-7列出了两个你应该了解的类。

表9-7 System.Collections.ObjectModel中有用的成员

System.Collections.ObjectModel中的类型	含 义
ObservableCollection<T>	表示能在添加、移除项或刷新整个列表时提供通知的动态数据集合。
ReadOnlyObservableCollection<T>	表示ObservableCollection<T>的只读版本

ObservableCollection<T>类十分有用，它可以在其内容发生变化时以某种方式通知外部对象（ReadOnlyObservableCollection<T>的操作与之类似，只不过是只读的）。

使用ObservableCollection<T>

创建一个新的名为FunWithObservableCollection的Console Application，在初始的C#代码文件中引入System.Collections.ObjectModel命名空间。在很多情况下，使用ObservableCollection<T>与使用List<T>完全相同，因为它们都实现了相同的核心接口。不同的是，ObservableCollection<T>实现了一个名为CollectionChanged的事件。该事件将在插入新项、移除（或重新分配）当前项或修改整个集合时触发。

与其他事件相同，CollectionChanged定义为委托（这里为NotifyCollectionChangedEventHandler）。该委托可以调用任何以object为第一个参数，以NotifyCollectionChangedEventArgs为第二个参数的方法。下面的Main()方法将生成一个包含Person对象的可观察集合，并绑定CollectionChanged事件：

```
class Program
{
  static void Main(string[] args)
  {
    //创建一个用来观察的集合，并添加一些Person对象
    ObservableCollection<Person> people = new ObservableCollection<Person>()
    {
      new Person{ FirstName = "Peter", LastName = "Murphy", Age = 52 },
      new Person{ FirstName = "Kevin", LastName = "Key", Age = 48 },
    };

    //绑定CollectionChanged事件
    people.CollectionChanged += people_CollectionChanged;
  }

  static void people_CollectionChanged(object sender,
    System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
  {
    throw new NotImplementedException();
  }
}
```

接下来的NotifyCollectionChangedEventArgs参数定义了两个重要属性，OldItems和NewItems，前者给出当前集合中事件触发之前的项，后者给出变化后的新项。不过，你可能只会在正常情况下查看列表。回想一下，插入、移除、重新定位或者重新设置项均会触发CollectionChanged事件。Action属性可用来测试NotifyCollectionChangedAction枚举的下列任意成员：

```
public enum NotifyCollectionChangedAction
{
  Add = 0,
  Remove = 1,
  Replace = 2,
  Move = 3,
  Reset = 4,
}
```

以下是CollectionChanged事件处理程序的一种实现，它在集合插入或移除项时打印旧的或新的集合：

```
static void people_CollectionChanged(object sender,
  System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
  //触发事件的行为是什么
  Console.WriteLine("Action for this event: {0}", e.Action);

  //是移除项
  if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Remove)
  {
    Console.WriteLine("Here are the OLD items:");
    foreach (Person p in e.OldItems)
    {
      Console.WriteLine(p.ToString());
    }
    Console.WriteLine();
  }

  //是添加项
  if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Add)
  {
    //显示添加后的新项
    Console.WriteLine("Here are the NEW items:");
    foreach (Person p in e.NewItems)
    {
      Console.WriteLine(p.ToString());
    }
  }
}
```

现在，假设我们更新了Main()方法来添加和移除一个项，你将看到类似下面的输出结果：

```
Action for this event: Add
Here are the NEW items:
Name:  Fred Smith, Age: 32

Action for this event: Remove
Here are the OLD items:
Name: Peter Murphy, Age: 52
```

以上就是.NET基础类库中以泛型为核心的命名空间。作为本章的结束，我们将学习如何构建自定义泛型方法和自定义泛型类型。

源代码 FunWithObservableCollection项目的源代码位于第9章的子目录下。

9.6 创建自定义泛型方法

尽管大多数开发者通常使用基础类库中已知的泛型类型，不过你也可以构建自己的泛型成员和自定义泛型类型。我们来看看如何在项目中添加自定义泛型。首先构建一个泛型包装方法。我们新建一个控制台应用程序CustomGenericMethods。

在构建自定义泛型方法时，你可以得到一个加强版的方法重载方式。在第2章中，我们知道重载可以为一个方法定义多个版本，它们以参数的数量和类型进行区分。

尽管在面向对象语言中重载是很有用的特性，但却经常发生多个方法实现相似功能的情况。例如，你需要构建一些方法，使用简单的规则交换两个数据。你可能会编写一个操作整数的新方法，如下所示：

```
// 交换两个整数
static void Swap(ref int a, ref int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

就这样还好。但如果还希望交换两个Person对象，就需要编写Swap()的新版本：

```
// 交换两个Person对象
static void Swap(ref Person a, ref Person b)
{
  Person temp;
  temp = a;
  a = b;
  b = temp;
}
```

毫无疑问，你知道这意味着什么。如果还需要交换浮点数、位图、汽车、按钮等，就需要构建更多的方法，这将给维护工作带来麻烦。你可以创建单一的（非泛型的）方法来操作object参数，但这将面对本章开始时所指出的所有问题，如装箱、拆箱、缺乏类型安全、显示转换等。

如果你要创建的方法重载只是输入参数不同，可以使用泛型。考虑如下的Swap<T>泛型方法，可以交换任意两个T：

```
// 该方法可以交换任意两个由类型参数<T>指定的项
static void Swap<T>(ref T a, ref T b)
{
  Console.WriteLine("You sent the Swap() method a {0}",
    typeof(T));
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

注意，在定义泛型方法时，类型参数在方法名称之后、参数列表之前进行指定。这里的Swap<T>()方法可以操作任意两个<T>类型的参数。为了更加直观地说明，我们还使用C#的typeof()操作符向控制台打印占位符的类型名称。现在考虑下面的Main()方法，可以交换整数和字符串：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Generic Methods *****\n");

  // 交换两个整数
  int a = 10, b = 90;
  Console.WriteLine("Before swap: {0}, {1}", a, b);
  Swap<int>(ref a, ref b);
  Console.WriteLine("After swap: {0}, {1}", a, b);
  Console.WriteLine();

  // 交换两个字符串
  string s1 = "Hello", s2 = "There";
  Console.WriteLine("Before swap: {0} {1}!", s1, s2);
  Swap<string>(ref s1, ref s2);
  Console.WriteLine("After swap: {0} {1}!", s1, s2);

  Console.ReadLine();
}
```

输出结果如下所示：

```
***** Fun with Custom Generic Methods *****

Before swap: 10, 90
You sent the Swap() method a System.Int32
After swap: 90, 10

Before swap: Hello There!
You sent the Swap() method a System.String
After swap: There Hello!
```

该方法最大的优势在于，只需要维护一个Swap<T>()版本，而且它能以类型安全的方式操作任意两个给定参数类型的项。更重要的是，栈数据保留在栈上，堆数据保留在堆上。

类型参数的推断

调用诸如Swap<T>之类的泛型方法时，当（且仅当）泛型方法需要参数时，我们可以选择省略类型参数，因为编译器会基于成员参数推断类型参数。举个例子，我们可以通过将以下代码添加到Main()中来交换两个System.Boolean类型的值：

```
// 编译器将推断System.Boolean
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

尽管编译器可以根据声明b1和b2的类型发现正确的类型参数，但你还是应该养成显式指定类型参数的习惯：

```
Swap<bool>(ref b1, ref b2);
```

这可以让你的同事很清楚地知道该方法是泛型的。另外，类型参数推断只在泛型方法至少有一个参数的时候起作用。例如，假设Program类包含如下的泛型方法：

```
static void DisplayBaseClass<T>()
{
  // BaseType是反射中使用的一个方法，反射将在第15章中介绍
  Console.WriteLine("Base class of {0} is: {1}.",
    typeof(T), typeof(T).BaseType);
}
```

这时，就必须在调用时提供类型参数：

```
static void Main(string[] args)
{
...
  // 如果方法没有参数，就必须提供类型参数
  DisplayBaseClass<int>();
  DisplayBaseClass<string>();

  // 编译器错误！没有参数时必须提供占位符
  // DisplayBaseClass();
  Console.ReadLine();
}
```

现在泛型方法Swap<T>和DisplayBaseClass<T>已经定义在应用程序的Program类中了。和其他方法一样，你可以在其他类类型（MyGenericMethods）中定义这些成员：

```
public static class MyGenericMethods
{
  public static void Swap<T>(ref T a, ref T b)
  {
    Console.WriteLine("You sent the Swap() method a {0}",
      typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
  }

  public static void DisplayBaseClass<T>()
  {
    Console.WriteLine("Base class of {0} is: {1}.",
      typeof(T), typeof(T).BaseType);
  }
}
```

因为Swap<T>和DisplayBaseClass<T>静态方法已被包含进这个全新的静类类型中，所以我们在调用任一成员时需要指定类型名称，例如：

```
MyGenericMethods.Swap<int>(ref a, ref b);
```

当然，泛型方法并不一定是静态的。如果Swap<T>和DisplayBaseClass<T>是实例方法（并且定义在非静态类中），那么简单创建一个MyGenericMethods的实例，然后使用该对象变量就可以调用它们：

```
MyGenericMethods c = new MyGenericMethods();
c.Swap<int>(ref a, ref b);
```

源代码 CustomGenericMethods项目的源代码位于Chapter 9子目录下。

9.7 创建自定义泛型结构和类

了解了如何定义和调用泛型方法之后，让我们将注意力转到泛型结构的构建上来（泛型类的构建过程是一样的），新建一个Console Application项目GenericPoint。假定我们构造了一个灵活的Point泛型结构：支持由一个类型参数来表示（X,Y）坐标的实际存储结构。调用者可以像下面这样创建Point<T>类型：

```
// 使用整数的Point
Point<int> p = new Point<int>(10, 10);

// 使用双精度数的Point
Point<double> p2 = new Point<double>(5.4, 3.3);
```

下面是Point<T>的完整定义，分析如下：

```
// 一个泛型Point结构
public struct Point<T>
{
  // 泛型状态数据
  private T xPos;
  private T yPos;

  // 泛型构造函数
  public Point(T xVal, T yVal)
  {
    xPos = xVal;
    yPos = yVal;
  }

  // 泛型属性
  public T X
  {
    get { return xPos; }
    set { xPos = value; }
  }

  public T Y
  {
    get { return yPos; }
    set { yPos = value; }
  }

  public override string ToString()
  {
    return string.Format("[{0}, {1}]", xPos, yPos);
  }

  // 重置字段为类型参数的默认值
  public void ResetPoint()
  {
    xPos = default(T);
    yPos = default(T);
  }
}
```

泛型代码中的default关键字

可见，Point<T>的类型参数出现在字段数据定义、构造函数参数和属性定义中。除此之外，请注意除了重写ToString()方法外，Point<T>在定义ResetPoint()方法时使用了一个新的语法：

```
// 在C#中，default关键字被重载
// 和泛型一起使用时，它表示一个类型参数的默认值
public void ResetPoint()
{
  X = default(T);
  Y = default(T);
}
```

在引入泛型时，default关键字被赋予了双重身份。除了在switch结构内部使用外，还可用于设置类型参数的默认值。这是非常有用的，因为一个泛型类型预先并不知道实际的占位符，因此无法安全地假设默认值是什么。类型参数的默认值如下：

- 数值的默认值为0；
- 引用类型的默认值为null；
- 一个结构的字段被设为0（值类型）或null（引用类型）。

对Point<T>而言，由于其调用者仅仅使用数值数据，所以可以直接把X和Y设为0。而使用default(T)语法，则可以提高泛型类型的灵活性。但无论如何，我们都可以像下面这样使用Point<T>：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Generic Structures *****\n");

  // 使用整数的Point
  Point<int> p = new Point<int>(10, 10);
  Console.WriteLine("p.ToString()={0}", p.ToString());
  p.ResetPoint();
  Console.WriteLine("p.ToString()={0}", p.ToString());
  Console.WriteLine();

  // 使用双精度数的Point
  Point<double> p2 = new Point<double>(5.4, 3.3);
  Console.WriteLine("p2.ToString()={0}", p2.ToString());
  p2.ResetPoint();
  Console.WriteLine("p2.ToString()={0}", p2.ToString());

  Console.ReadLine();
}
```

输出结果如下：

```
***** Fun with Generic Structures *****

p.ToString()=[10, 10]
p.ToString()=[0, 0]

p2.ToString()=[5.4, 3.3]
p2.ToString()=[0, 0]
```

源代码　GenericPoint项目的源代码位于Chapter 9子目录下。

9.8 类型参数的约束

正如本章所描述的，任何泛型项都必须至少有一个类型参数，并在与泛型类型或参数交互时指定该类型参数。这可以使我们构建类型安全的代码。而.NET平台使用where关键字可以得到更加具体的类型参数信息。

使用这个关键字，可以对给定的类型参数添加一组约束，C#编译器将在编译时检查这些约束。可对类型参数添加的约束如表9-8所示。

表9-8　泛型类型参数的约束

泛型约束	作　用
where T : struct	该类型参数<T>必须在其继承链中包含System.ValueType值类型，即必须为结构
where T : class	该类型参数<T>不能在其继承链中包含System.ValueType值类型，即<T>必须是引用类型
where T : new()	该类型参数<T>必须包含一个默认的构造函数。因为无法预知自定义构造函数的格式，所以如果泛型类型必须创建一个类型参数的实例，这将是非常有用的。注意在有多个约束的类型上，此约束必须列在末尾
where T : NameOfBaseClass	该类型参数<T>必须派生于NameOfBaseClass指定的类
where T : NameOfInterface	该类型参数<T>必须实现NameOfInterface指定的接口，多个接口必须用逗号隔开

除非要创建类型安全的自定义集合，否则你很少会在C#项目中使用where关键字。下面的部分代码示例演示了如何使用where关键字。

9.8.1 使用where关键字的示例

假设你已经创建了一个自定义泛型类，并且希望确保类型参数包含一个默认的构造函数。这在自定义泛型类需要创建T的实例时非常有用，因为对很多常用类型来说，默认构造函数是唯一的构造函数。同样，对T进行这种约束可以获得编译时检查。如果T是引用类型，程序员会重新定义在类中定义的默认构造函数（在定义自己的类时，可能移除了默认的构造函数）：

```
// MyGenericClass派生自object，而包含项必须拥有默认的构造函数
public class MyGenericClass<T> where T : new()
{
  ...
}
```

注意，where子句指定类型参数需遵循某种约束，在其之后为冒号操作符。冒号后面列出了各个可能的约束（本例为默认构造函数）。下面是另一个示例：

```
// MyGenericClass派生自object，并且包含项为实现了IDrawable的类，并且必须支持默认构造函数
public class MyGenericClass<T> where T : class, IDrawable, new()
```

```
{
  ...
}
```

本例中的T包含3个条件。它必须为引用类型（不能为结构），因为使用了class标记。其次，T必须实现IDrawable接口。最后，它必须包含默认的构造函数。这些约束组成了用逗号分隔的列表。但要注意的是，new()约束必须总是在列表的末尾。因此，如下的代码无法编译：

```
// 错误！new()约束必须在最后
public class MyGenericClass<T> where T : new(), class, IDrawable
{
  ...
}
```

如果你创建的自定义泛型集合指定了多个类型参数，可以为每个类型参数指定约束集，各约束集之间用where子句分隔：

```
// <K>必须扩展SomeBaseClass，并且必须包含默认构造函数
// <T>必须为结构，并且实现泛型IComparable接口
public class MyGenericClass<K, T> where K : SomeBaseClass, new()
    where T : struct, IComparable<T>
{
  ...
}
```

尽管构建完整的自定义泛型集合类的情况很少见，但你还可以将where关键字用于泛型方法。例如，如果要指定泛型Swap<T>方法只能操作结构，可以这样更新代码：

```
// 该方法可以交换任何结构，但不能用于类
static void Swap<T>(ref T a, ref T b) where T : struct
{
  ...
}
```

注意，如果要以这种方式约束Swap()方法，就不能再对string对象进行交换了（如示例代码所示），因为string为引用类型。

9.8.2 操作符约束的不足

在本章即将结束的时候，我还想再评价一下泛型方法和约束。在创建泛型方法时，如果对类型参数应用任何C#操作符（+、-、*、==等），都将产生令人惊奇的编译器错误。举例来说，想象一下可以对泛型类型进行Add()、Subtract()、Multiply()和Divide()的类：

```
// 编译器错误！不能对类型参数使用操作符
public class BasicMath<T>
{
  public T Add(T arg1, T arg2)
  { return arg1 + arg2; }
  public T Subtract(T arg1, T arg2)
  { return arg1 - arg2; }
  public T Multiply(T arg1, T arg2)
  { return arg1 * arg2; }
  public T Divide(T arg1, T arg2)
  { return arg1 / arg2; }
}
```

9

遗憾的是，上面的BasicMath类并不能编译。请记住泛型并不是万能的，不能对类型参数进行操作符操作看上去就是它的主要限制之一。当然，数值数据可以使用C#的二进制操作符。但如果<T>是一个自定义类或结构类型，编译器就不能保证它已经重载了+、-、*和/操作符。理想情况下，C#应当允许泛型类型支持操作符约束，例如：

```
// 演示代码
public class BasicMath<T> where T : operator +, operator -,
  operator *, operator /
{
  public T Add(T arg1, T arg2)
  { return arg1 + arg2; }
  public T Subtract(T arg1, T arg2)
  { return arg1 - arg2; }
  public T Multiply(T arg1, T arg2)
  { return arg1 * arg2; }
  public T Divide(T arg1, T arg2)
  { return arg1 / arg2; }
}
```

唉，当前的C#版本还不支持操作符约束。但我们可以定义支持这些操作符的接口（C#接口可以定义操作符），然后为泛型类指定该接口的约束来满足这一需求（但工作量较大）。不管怎样，本章完成了关于构建自定义泛型类型的介绍。在下一章介绍.NET委托类型时，我们还将讨论泛型。

9.9　小结

本章首先介绍了System.Collections和System.Collections.Specialized中的非泛型集合类型，以及很多非泛型容器中存在的问题，如缺乏类型安全和装箱拆箱操作带来的运行时开销。正因为如此，如今的.NET程序通常使用System.Collections.Generic和System.Collections.ObjectModel中的泛型集合类。

我们已经看到，使用泛型能够在创建对象时（在泛型方法中为调用对象时）才指定类型的占位符（也就是类型参数）。从本质上来说，泛型解决了.NET 1.1开发中烦人的装箱和类型安全问题。此外，泛型类型一般不需要构建自定义集合类型。

最后要说明的是，在.NET基础类库中泛型随处可见。本章关注的是泛型集合。然而在你阅读本书剩余章节（或深入研究.NET平台）时，你会发现在一些命名空间下还有泛型类、泛型结构和泛型委托，而且在非泛型类中还可以存在泛型成员。

第 10 章

委托、事件和Lambda表达式

截止到现在，本书讲解的每个应用程序，从某种意义上说，都在给Main()方法添加向指定对象发送请求的各种代码段。但是，很多应用程序需要对象使用某种回调机制，能够与创建它的实体进行通信。尽管回调机制可用于各种应用程序，但它们对于图形用户界面来说尤其重要，因为控件（如按钮）需要在正确的环境下调用外部方法（如单击按钮时、鼠标滑过按钮表面时等）。

在.NET平台下，委托类型用来定义和响应应用程序中的回调。事实上，.NET委托类型是一个类型安全的对象，指向可以以后调用的其他方法。和传统的C++函数指针不同，.NET委托是内置支持多路广播和异步方法调用的对象。

在本章中，我们将学会如何创建与应用委托类型，接下来再研究C#中的event关键字，它使我们处理委托类型的过程更加简化和高效。最后，本章讨论C#中与委托和事件相关的语言新特性，包括匿名方法和方法组转换。

本章最后会讨论Lambda表达式。使用新的Lambda操作符（=>），我们就可以在任何需要强类型委托的情况下指定一个代码语句（以及传入这个代码语句的参数）。可以看到，Lambda表达式只是匿名方法的一种伪装，提供了一种简单的方式来使用委托。

10.1 .NET 委托类型

在正式定义.NET委托之前，让我们先来了解一下背景。历史上，Windows API经常使用C语言风格的函数指针来创建称为回调函数或简称为回调[①]的实体。使用回调，程序员可以使一个函数返回报告给（即回调）程序中的另一个函数。使用这种方法，Windows开发者可以处理按钮单击、鼠标移动、菜单选择以及内存中两个实体间的双向通信。

标准C语言风格回调函数的问题在于，它们除了原始内存地址外无法表示其他信息。而理想中的回调应该包含更多类型安全信息，例如参数的数量与类型、所指向的方法的返回值（如果有的话）。很可惜，这些传统的回调功能都做不到，同时，正如有些读者可能已经想到的，它们因此经常成为bug、崩溃和其他运行时灾难的源头。尽管如此，回调还是非常有用的实体。

在.NET Framework里，回调仍是可能的，它们的功能是由使用更为安全和面向对象的委托

---

① 回调（callback）一词本身指的是可以作为参数传给其他代码的一段可执行代码。普通调用往往是高层代码（如应用程序）去调用处在低层的函数（如系统函数、库函数）。而回调时，则是在低层函数执行时调用高层的代码，此术语由此得名。回调（通过委托实现）是多态和泛型编程（见第9章）之外的替代方案。——编者注

（delegate）来完成的。本质上来讲，委托是一个类型安全的对象，它指向程序中另一个以后会被调用的方法（或多个方法）。委托类型包含3个重要的信息：

- 它所调用的方法的名称；
- 该方法的参数（可选）；
- 该方法的返回值类型（可选）。

说明　.NET委托既可以指向静态方法，也可以指向实例方法。

当一个委托对象被创建并提供了上述信息后，它可以在运行时动态调用其指向的方法。可以看到，.NET Framework中每个委托（包括自定义委托）都被自动赋予同步或异步访问方法的能力，可以不用手工创建与管理一个Thread对象而直接调用另一个辅助执行线程上的方法，这大大简化了编程工作。

说明　我们将在第19章研究线程和异步调用时介绍委托类型的异步行为。本章我们只关注委托类型的同步性质。

10.1.1　在C#中定义委托类型

在C#中创建一个委托类型时，需要使用delegate关键字。委托的名称可以自由选择。不过，必须定义委托来匹配它指向的方法的签名。例如，假定我们要创建一个名为BinaryOp的委托，它可以指向任何输入两个整数返回一个整数的方法：

```
// 这个委托可以指向任何传入两个整数返回一个整数的方法
public delegate int BinaryOp(int x, int y);
```

当C#编译器处理委托类型时，它先自动产生一个派生自System.MulticastDelegate的密封类。这个类与它的基类System.Delegate一起为委托提供必要的基础设施，以维护[①]以后将要调用方法的列表。例如，如果我们通过ildasm.exe来查看BinaryOp委托，将发现如图10-1所示的各项。

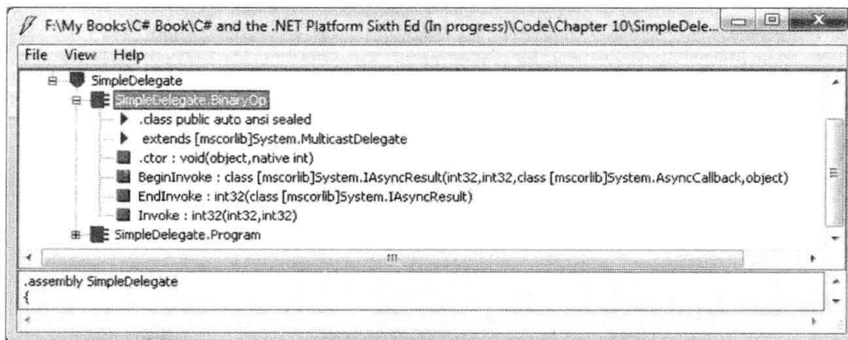


图10-1　C# delegate关键字呈现为一个派生自System.MulticastDelegate的密封类

① 此处和下面的委托维护的方法都指委托所指向的方法。——编者注

可以看到，生成的BinaryOp类定义了3个公共方法。Invoke()可能是核心方法，因为它被用来以同步方式调用委托对象维护的每个方法。这里的同步就是指调用者必须等待调用完成才能继续执行。奇怪的是，同步Invoke()方法不能在C#中直接调用。稍后我们将看到，当使用适当的C#语法的时候，Invoke()将在后台调用。

BeginInvoke()和EndInvoke()方法能在第二个执行线程上异步调用当前方法。如果读者有多线程背景知识，会明白开发人员创建第二个执行线程的一个最常见的原因，是调用比较耗时的方法。尽管.NET基础类库为多线程和并行编程专门提供了整个命名空间（System.Threading），而委托顺带就提供了这一功能。

那么，编译器又是如何确切知道怎样定义Invoke()、BeginInvoke()和EndInvoke()方法的呢？为了理解这个过程，让我们看看下面这段代码，它说明了BinaryOp类的症结所在（其中**粗体**部分标记出定义的委托类型指定的项）：

```
sealed class BinaryOp : System.MulticastDelegate
{
  public int Invoke(int x, int y);
  public IAsyncResult BeginInvoke(int x, int y,
    AsyncCallback cb, object state);
  public int EndInvoke(IAsyncResult result);
}
```

首先，请注意由Invoke()方法定义的参数和返回值完全匹配BinaryOp委托的定义。BeginInvoke()成员前面的参数（这里是两个整数）也基于BinaryOp委托；但BeginInvoke()方法将总是提供最后两个参数（AsyncCallback类型与object类型），用于异步方法调用。最后，EndInvoke()方法的返回值与初始的委托声明相同，总是以一个实现了IAsyncResult接口的对象作为其唯一的参数。

让我们看看另一个例子。假设读者定义了一个可以指向任何方法、返回1个字符串、接受3个布尔输入参数的委托类型：

```
public delegate string MyDelegate(bool a, bool b, bool c);
```

这一次，自动产生的类分解如下：

```
sealed class MyDelegate : System.MulticastDelegate
{
  public string Invoke(bool a, bool b, bool c);
  public IAsyncResult BeginInvoke(bool a, bool b, bool c,
    AsyncCallback cb, object state);
  public string EndInvoke(IAsyncResult result);
}
```

委托还可以指向包含任意数量out或ref参数（以及用params关键字标记的数组参数）的方法。例如，假设有委托类型如下：

```
public delegate string MyOtherDelegate(out bool a, ref bool b, int c);
```

Invoke()与BeginInvoke()方法的签名将不出所料，但EndInvoke()方法稍有变化，其中包括委托类型定义的所有的out/ref参数：

```
public sealed class MyOtherDelegate : System.MulticastDelegate
{
  public string Invoke(out bool a, ref bool b, int c);
  public IAsyncResult BeginInvoke(out bool a, ref bool b, int c,
    AsyncCallback cb, object state);
```

```
  public string EndInvoke(out bool a, ref bool b, IAsyncResult result);
}
```

综上所述，C#委托类型定义会生成一个密封类，它含有3个编译器生成的方法，这3个方法的参数与返回值基于委托的声明。下面这段伪代码说明了这个基本模式：

```
// 这只是伪代码
public sealed class DelegateName : System.MulticastDelegate
{
  public delegateReturnValue Invoke(allDelegateInputRefAndOutParams);

  public IAsyncResult BeginInvoke(allDelegateInputRefAndOutParams,
    AsyncCallback cb, object state);

  public delegateReturnValue EndInvoke(allDelegateRefAndOutParams,
    IAsyncResult result);
}
```

10.1.2　System.MulticastDelegate与System.Delegate基类

使用C#中delegate关键字创建委托的时候，也就间接声明了一个派生自System.MulticastDelegate的类。这个类使其继承类可以访问包含由委托对象维护的方法地址的列表以及一些处理调用列表的附加方法（与少数重载的操作符）。下面是System.MulticastDelegate的部分成员：

```
public abstract class MulticastDelegate : Delegate
{
  // 返回所指向的方法列表
  public sealed override Delegate[] GetInvocationList();

  // 重载的操作符
  public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
  public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);

  // 用来在内部管理委托所维护的方法列表
  private IntPtr _invocationCount;
  private object _invocationList;
}
```

System.MulticastDelegate从它的父类System.Delegate继承了更多功能。下面是父类的部分定义：

```
public abstract class Delegate : ICloneable, ISerializable
{
  // 与函数列表交互的方法
  public static Delegate Combine(params Delegate[] delegates);
  public static Delegate Combine(Delegate a, Delegate b);
  public static Delegate Remove(Delegate source, Delegate value);
  public static Delegate RemoveAll(Delegate source, Delegate value);

  // 重载的操作符
  public static bool operator ==(Delegate d1, Delegate d2);
  public static bool operator !=(Delegate d1, Delegate d2);

  // 扩展委托目标的属性
  public MethodInfo Method { get; }
  public object Target { get; }
}
```

要记住，我们永远不会直接派生自这些基类（如果这样做的话，会有编译器错误）。然而，如果我们使用delegate关键字，就间接创建一个类，这个类“是”MulticastDelegate。表10-1列举了所有委托类型都共有的核心成员。

表10-1 System.MulticastDelegate/System.Delegate部分成员

继承成员	作　用
Method	此属性返回System.Reflection.MethodInfo对象，用以表示委托维护的静态方法的详细信息
Target	如果方法调用是定义在对象级别的（而不是静态方法），Target返回表示委托维护的方法的对象。如果Target返回值为null，调用的方法是一个静态成员
Combine()	此静态方法给委托维护的列表添加一个方法。在C#中，使用重载+=操作符作为简化符号调用此方法
GetInvocationList()	此方法返回一个System.Delegate类型的数组，其中数组中的每个元素都表示一个可调用的特定方法
Remove() RemoveAll()	这些静态方法从调用列表中移除一个（或所有）方法。在C#中，Remove()方法可通过使用重载-=操作符来调用

10.2 最简单的委托示例

初次接触委托可能感觉很难。不要紧张，接下来，让我们看一个非常简单的使用BinaryOp委托的示例，我们以前见过，这是一个名为SimpleDelegate的控制台应用程序项目。下面是完整的代码和代码分析：

```
namespace SimpleDelegate
{
  // 这个委托可以指向任何传入两个整数并返回一个整数的方法
  public delegate int BinaryOp(int x, int y);

  // 这个类包含了BinaryOp将指向的方法
  public class SimpleMath
  {
    public static int Add(int x, int y)
    { return x + y; }
    public static int Subtract(int x, int y)
    { return x - y; }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Simple Delegate Example *****\n");

      // 创建一个指向SimpleMath.Add()方法的BinaryOp对象
      BinaryOp b = new BinaryOp(SimpleMath.Add);

      // 使用委托对象间接调用Add()方法
      Console.WriteLine("10 + 10 is {0}", b(10, 10));
      Console.ReadLine();
    }
  }
}
```

还要注意的是BinaryOp委托类型声明的格式，它指向任何一个带有两个整数参数并返回一个整数的方法。因此，我们可以创建一个名为SimpleMath的类，其中定义了两个完全匹配BinaryOp委托定义模式的静态方法。

如果要将目标对象方法插入指定委托对象，只要向委托的构造函数传入方法名称即可。

```
// 创建指向SimpleMath.Add()的BinaryOp委托对象
BinaryOp b = new BinaryOp(SimpleMath.Add);
```

这时，我们可以使用类似直接函数调用的语法调用指向成员：

```
// Invoke()在这里被调用了
Console.WriteLine("10 + 10 is {0}", b(10, 10));
```

在底层，运行库实际上在MulticastDelegate派生类上调用了编译器生成的Invoke()方法。读者可以通过在ildasm.exe中打开程序集，观察Main()函数中的CIL代码自己进行验证：

```
.method private hidebysig static void Main(string[] args) cil managed
{
...
  callvirt instance int32 SimpleDelegate.BinaryOp::Invoke(int32, int32)
}
```

尽管C#不需要我们在代码库中显式调用Invoke()，但是我们也完全可以这么做。因此，如下代码语句是可行的：

```
Console.WriteLine("10 + 10 is {0}", b.Invoke(10, 10));
```

.NET委托是类型安全的。所以如果读者试图将一个不“匹配模式”的方法传入委托，将会收到编译器错误。例如，假定SimpleMath类定义了另外一个名为SquareNumber()的方法（它的输入参数为整数）：

```
public class SimpleMath
{
...
  public static int SquareNumber(int a)
  { return a * a; }
}
```

由于BinaryOp委托仅可指向带有两个整数参数并返回一个整数的方法，下面这段代码是非法的，将无法编译：

```
// 编译器错误！方法不匹配委托的模式
BinaryOp b2 = new BinaryOp(SimpleMath.SquareNumber);
```

委托对象

让我们在Program类中创建一个名为DisplayDelegateInfo()的静态方法来丰富当前的示例。这个方法将输出由传入的委托类型所维护的方法的名称和定义该方法的类的名称。我们通过迭代由GetInvocationList()返回的System.Delegate数组，调用每个对象的Target和Method属性：

```
static void DisplayDelegateInfo(Delegate delObj)
{
  // 输出委托调用列表中每个成员的名称
  foreach (Delegate d in delObj.GetInvocationList())
  {
```

```
    Console.WriteLine("Method Name: {0}", d.Method);
    Console.WriteLine("Type Name: {0}", d.Target);
  }
}
```

假定你已经修改了Main()方法来调用这个新的辅助方法：

```
BinaryOp b = new BinaryOp(SimpleMath.Add);
DisplayDelegateInfo(b);
```

将会得到如下所示的输出结果：

```
***** Simple Delegate Example *****

Method Name: Int32 Add(Int32, Int32)
Type Name:
10 + 10 is 20
```

请注意SimpleMath类型的名称现在并没有被Target属性显示。这是因为我们的BinaryOp委托指向静态方法，所以没有对象被引用！而如果修改Add()和Subtract()方法为非静态的（删除static关键字），将能创建SimpleMath类的一个实例，并按照如下所示通过对象引用来指定调用的方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Delegate Example *****\n");

  // .NET委托也可以指向实例方法
  SimpleMath m = new SimpleMath();
  BinaryOp b = new BinaryOp(m.Add);

  // 显示这个对象的信息
  DisplayDelegateInfo(b);

  Console.WriteLine("10 + 10 is {0}", b(10, 10));
  Console.ReadLine();
}
```

这样，我们就能得到如下所示的输出结果：

```
***** Simple Delegate Example *****

Method Name: Int32 Add(Int32, Int32)
Type Name: SimpleDelegate.SimpleMath
10 + 10 is 20
```

源代码　SimpleDelegate项目的源代码位于Chapter 10子目录下。

10.3　使用委托发送对象状态通知

很明显，之前的SimpleDelegate示例是纯粹用来说明委托作用的，因为仅为了加两个数创建一个

10

委托没有多大必要。为了提供更现实的委托应用，我们使用委托来定义Car类，它可以通知外部实体当前引擎的状态。下面是我们要进行的步骤。

(1) 定义将通知发送给调用者的委托类型。

(2) 声明Car类中每个委托类型的成员变量。

(3) 在Car上创建辅助函数使调用者能指定由委托成员变量保存的方法。

(4) 修改Accelerate()方法以在适当情形下调用委托的调用列表。

首先，新建一个叫CarDelegate的控制台应用程序。现在，按如下所示定义Car类：

```
public class Car
{
  // 内部状态数据
  public int CurrentSpeed { get; set; }
  public int MaxSpeed { get; set; }
  public string PetName { get; set; }

  // 汽车能用还是不能用
  private bool carIsDead;

  // 类构造函数
  public Car() { MaxSpeed = 100; }
  public Car(string name, int maxSp, int currSp)
  {
    CurrentSpeed = currSp;
    MaxSpeed = maxSp;
    PetName = name;
  }
}
```

现在，考虑如下对Car类的更新，它强调了前面3点：

```
public class Car
{
  ...
  // 1) 定义委托类型
  public delegate void CarEngineHandler(string msgForCaller);

  // 2) 定义每个委托类型的成员变量
  private CarEngineHandler listOfHandlers;

  // 3) 向调用者添加注册函数
  public void RegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    listOfHandlers = methodToCall;
  }
}
```

注意，本例中我们直接在Car类作用域里定义委托类型。浏览基础类库时会发现，将委托定义在使用它的类型作用域里是很普遍的。委托类型CarEngineHandler指向的方法只有一个字符串型的输入参数，返回值为void。

下一步，注意我们声明了一个私有成员变量（名为listOfHandlers）和一个辅助函数（RegisterWith-CarEngine()），从而使客户端能给委托调用列表添加方法。

说明 严格地说，我们可以将委托成员变量定义为公共的，这样就不需要创建额外的注册方法。然而，如果将委托成员定义为私有的，我们就强制了封装服务并提供了更类型安全的解决方案。在研究C# event关键字的时候，我们会再次讨论公共委托成员变量的风险。

在这里，我们需要创建Accelerate()方法。记住，这里我们需要使Car对象向订阅者发送引擎相关的消息。更新如下：（代码也有不同）

```
// 4) 实现Accelerate()方法以在某些情况下调用委托的调用列表
//    如果汽车不能用了，触发引爆事件
public void Accelerate(int delta)
{
  // If this car is "dead," send dead message.
  if (carIsDead)
  {
    if (listOfHandlers != null)
      listOfHandlers("Sorry, this car is dead...");
  }
  else
  {
    CurrentSpeed += delta;

    // 快不能用了吗
    if (10 == (MaxSpeed - CurrentSpeed)
        && listOfHandlers != null)
    {
      listOfHandlers("Careful buddy! Gonna blow!");
    }

    if (CurrentSpeed >= MaxSpeed)
      carIsDead = true;
    else
      Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
  }
}
```

要注意在我们调用listOfHandlers成员变量保存的方法之前，需要检查该变量是否是空值。原因在于通过调用RegisterWithCarEngine()辅助方法分配这些对象是调用者的任务。如果调用者没有调用这个方法而我们试图调用委托调用列表，将在运行时触发一个引用为空异常（NullReferenceException）并使程序失败（很明显，那非常糟糕）！现在我们有了委托构造，下面来看修改后的Program类：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Delegates as event enablers *****\n");

    // 首先，创建一个Car对象
    Car c1 = new Car("SlugBug", 100, 10);

    // 现在，告诉汽车，它想要向我们发送信息时调用哪个方法
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

    // 加速（这将触发事件）
    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
```

```
      c1.Accelerate(20);
    Console.ReadLine();
  }

  // 要传入事件的方法
  public static void OnCarEngineEvent(string msg)
  {
    Console.WriteLine("\n***** Message From Car Object *****");
    Console.WriteLine("=> {0}", msg);
    Console.WriteLine("***********************************\n");
  }
}
```

Main()方法首先新建了一个Car对象。由于我们对引擎事件感兴趣，所以下一步调用自定义的注册函数RegisterWithCarEngine()。注意，该方法要求传入一个内嵌的CarEngineHandler委托的实例，与其他委托一样，我们指定一个“所指向的方法”作为构造函数的参数。该示例的关键在于，所讨论的方法位于Program类的内部。并且OnCarEngineEvent()方法与相关的委托完全匹配，也包含string类型的输入参数和void返回类型。考虑如下所示的输出结果：

```
***** Delegates as event enablers *****

***** Speeding up *****
CurrentSpeed = 30
CurrentSpeed = 50
CurrentSpeed = 70

***** Message From Car Object *****
=> Careful buddy!  Gonna blow!
***********************************

CurrentSpeed = 90

***** Message From Car Object *****
=> Sorry, this car is dead...
***********************************
```

10.3.1 支持多路广播

回想一下，.NET委托内置支持多路广播。换句话说，一个委托对象可以维护一个可调用方法的列表而不只是单独一个方法。给一个委托对象添加多个方法时，不用直接分配，重载+=操作符即可。为使Car类支持多路广播，可以修改RegisterWithCarEngine方法，具体如下所示：

```
public class Car
{
  // 现在支持多路广播
  // 注意现在我们正在使用+=操作符，而不是赋值操作符 (=)
  public void RegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    listOfHandlers += methodToCall;
  }
...
}
```

对委托对象使用+=操作符时，编译器将该问题转换为一个对静态Delegate.Combine()方法的调用（事实上，也可以直接调用Delegate.Combine()方法，不过使用+=操作符是更简洁的选择）。思考一下RegisterWithCarEngine()方法的CIL实现：

```
public void RegisterWithCarEngine( CarEngineHandler methodToCall )
{
  if (listOfHandlers == null)
    listOfHandlers = methodToCall;
  else
    Delegate.Combine(listOfHandlers, methodToCall);
}
```

这样，调用者就可以为同样的回调注册多个目标对象了。这里，第二个处理程序以大写形式打印传入的消息，以供显示：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Delegates as event enablers *****\n");

    // 首先，创建Car对象
    Car c1 = new Car("SlugBug", 100, 10);

    // 为通知注册多个目标
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent2));

    // 加速（这将触发该事件）
    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);
    Console.ReadLine();
  }

  // 现在在发送通知信息时，Car类将调用两个方法
  public static void OnCarEngineEvent(string msg)
  {
    Console.WriteLine("\n***** Message From Car Object *****");
    Console.WriteLine("=> {0}", msg);
    Console.WriteLine("***********************************\n");
  }

  public static void OnCarEngineEvent2(string msg)
  {
    Console.WriteLine("=> {0}", msg.ToUpper());
  }
}
```

10.3.2 从委托的调用列表中移除成员

Delegate类还定义了一个静态Remove()方法，允许调用者动态地从委托对象的调用列表中移除方法。这样，调用者就可以在运行时简单地“退订”某个已知的通知。你可以直接在代码中调用Delegate.Remove()，不过C#开发者可以使用-=操作符作为简写方式。为Car类添加新的方法，允许调用者从调用列表中移除某个方法：

```
public class Car
{
...
  public void UnRegisterWithCarEngine(CarEngineHandler methodToCall)
  {
    listOfHandlers -= methodToCall;
  }
}
```

这样改完Car类之后，再按如下代码修改Main()，就可以让第二个处理程序停止接收引擎通知：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Delegates as event enablers *****\n");

  // 首先，创建Car对象
  Car c1 = new Car("SlugBug", 100, 10);
  c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

  // 先绑定委托对象，稍后再注销
  Car.CarEngineHandler handler2 = new Car.CarEngineHandler(OnCarEngineEvent2);
  c1.RegisterWithCarEngine(handler2);

  // 加速（将触发事件）
  Console.WriteLine("***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);

  // 注销第二个处理程序
  c1.UnRegisterWithCarEngine(handler2);

  // 看不到大写的消息了
  Console.WriteLine("***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);

  Console.ReadLine();
}
```

Main()方法中的一处不同是，我们创建了Car.CarEngineHandler对象，并将它保存在本地变量中，这样就可以使用该对象在后面注销通知。在第二次对Car对象进行加速的时候，就看不到大写的传入消息了，因为我们已经在委托列表中移除了这个目标。

源代码 CarDelegate项目的源代码位于Chapter 10子目录下。

10.3.3 方法组转换语法

在CarDelegate这个示例中，我们显式地创建了Car.CarEngineHandler委托对象的实例，以注册和注销引擎通知：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Delegates as event enablers *****\n");

  Car c1 = new Car("SlugBug", 100, 10);
```

```
    c1.RegisterWithCarEngine(new Car.CarEngineHandler(OnCarEngineEvent));

    Car.CarEngineHandler handler2 =
      new Car.CarEngineHandler(OnCarEngineEvent2);
    c1.RegisterWithCarEngine(handler2);
  ...
  }
```

当然，如果要调用MulticastDelegate或Delegate中继承的任何成员，手工创建一个委托变量是最直接的方式。但大多数情况下，我们并不需要依靠委托对象。我们常常使用委托对象只是为了传递作为构造函数参数的方法名称。

为了简化操作，C#提供了一种叫做方法组转换的简便方法。该特性允许我们在调用以委托作为参数的方法时直接提供方法的名称，而不用创建委托对象。

说明 本章稍后将使用方法组转换语法简化C#事件的注册。

为了演示这一特性，新建一个Console Application，取名为CarDelegateMethodGroupConversion，并插入在CarDelegate项目中定义的Car类。现在，考虑下面的Program类，使用方法组转换来注册和注销引擎通知：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Method Group Conversion *****\n");
    Car c1 = new Car();

    //注册简单的方法名称
    c1.RegisterWithCarEngine(CallMeHere);

    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);

    //注销简单的方法名称
    c1.UnRegisterWithCarEngine(CallMeHere);

    // 没有通知
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);

    Console.ReadLine();
  }

  static void CallMeHere(string msg)
  {
    Console.WriteLine("=> Message from Car: {0}", msg);
  }
}
```

注意，我们没有直接分配相关的委托对象，而是简单地指定了与委托期望的签名相匹配的方法（返回void并且以string为参数）。要知道C#编译器仍然能确保类型安全。因此，如果CallMeHere()方法没有string参数或没有返回void，都将得到编译器错误。

源代码　CarDelegateMethodGroupConversion项目的源代码位于Chapter 10子目录下。

10.4 泛型委托

第9章提到过，C#允许我们定义泛型委托类型。例如，假设我们希望定义一个委托类型来调用任何返回void并且接受单个参数的方法。如果这个参数可能会不同，我们就可以通过类型参数来构建。为了演示，请考虑GenericDelegate新控制台应用程序中的如下代码：

```
namespace GenericDelegate
{
  // 这个泛型委托可以调用任何返回void并接受单个参数的方法
  public delegate void MyGenericDelegate<T>(T arg);

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Generic Delegates *****\n");

      // 注册目标
      MyGenericDelegate<string> strTarget =
        new MyGenericDelegate<string>(StringTarget);
      strTarget("Some string data");

      MyGenericDelegate<int> intTarget =
        new MyGenericDelegate<int>(IntTarget);
      intTarget(9);
      Console.ReadLine();
    }

    static void StringTarget(string arg)
    {
      Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
    }

    static void IntTarget(int arg)
    {
      Console.WriteLine("++arg is: {0}", ++arg);
    }
  }
}
```

注意，MyGenericDelegate<T>定义了一个类型参数来表示要传入委托目标的实参。在创建这个类型实例时，我们需要指定类型参数的值以及委托将调用的方法的名称。因此，如果指定了字符串类型，我们就可以把字符串值传入目标方法：

```
// 创建MyGenericDelegate<T>的实例，使用字符串作为类型参数
MyGenericDelegate<string> strTarget =
  new MyGenericDelegate<string>(StringTarget);
strTarget("Some string data");
```

由于strTarget对象的格式，StringTarget()方法现在必须接受一个字符串作为参数：

```
static void StringTarget(string arg)
{
  Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}
```

源代码 GenericDelegate项目的源代码位于Chapter 10子目录下。

泛型Action<>和Func<>委托

我们已经从本章看到，使用委托在应用程序中进行回调时需要遵循以下步骤：

- 自定义一个与要指向的方法的格式相匹配的委托；
- 创建自定义委托的实例，将方法名作为构造函数的参数；
- 通过调用委托对象的Invoke()来间接调用该方法。

采用这种方法通常会构造大量只会用于当前任务的自定义委托（如MyGenericDelegate<T>、CarEngineHandler等）。尽管有时候需要为项目定义一些具备独特名称的委托，但也有时候委托名无关紧要。许多情况下，我们只需要接受一组参数并返回一个值（或void）的委托。这时候，我们可以使用框架内置的Action<>和Func<>委托。为了演示它们的用途，新建一个名为ActionAndFuncDelegates的Console Application项目。

泛型Action<>委托定义在mscorlib.dll和System.Core.dll中的System命名空间中。它们可以指向多至16个参数（应该够了吧！）并返回void的方法。由于Action<>是泛型委托，因此还需要指定各个参数的基础类型。

更新Program类，新建一个接受3个不同参数的静态方法：

```
// Action<>委托的一个目标
static void DisplayMessage(string msg, ConsoleColor txtColor, int printCount)
{
  // 设置命令行文本的颜色
  ConsoleColor previous = Console.ForegroundColor;
  Console.ForegroundColor = txtColor;

  for (int i = 0; i < printCount; i++)
  {
    Console.WriteLine(msg);
  }

  // 重置颜色
  Console.ForegroundColor = previous;
}
```

现在，要把程序流传递给DisplayMessage()方法，我们可以使用Action<>委托，而不必手工构建自定义委托，如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Action and Func *****");

  // 使用Action<>委托来指向DisplayMessage
  Action<string, ConsoleColor, int> actionTarget =
```

```
    new Action<string, ConsoleColor, int>(DisplayMessage);
  actionTarget("Action Message!", ConsoleColor.Yellow, 5);

  Console.ReadLine();
}
```

如你所见，使用Action<>委托避免了创建自定义委托的烦恼。但它只能指向返回void的方法。如果要指向具有返回值的方法（同时又不想编写自定义委托），可以使用Func<>。

泛型Func<>委托可以指向多至16个参数（和Action<>一样）并具有自定义返回值的方法。为了演示，在Program类中添加如下所示的新方法：

```
// Func<>委托的目标
static int Add(int x, int y)
{
    return x + y;
}
```

本章前面构建了一个自定义的BinaryOp委托，用来指向加减法方法。不过，我们可以使用包含3个类型参数的Func<>版本来简化我们的工作。注意Func<>的最后一个类型参数总是方法的返回值。为了巩固这一点，假设Program类还定义了下面的方法：

```
static string SumToString(int x, int y)
{
  return (x + y).ToString();
}
```

现在，Main()方法可以调用这两个方法，如下所示：

```
Func<int, int, int> funcTarget = new Func<int, int, int>(Add);
int result = funcTarget.Invoke(40, 40);
Console.WriteLine("40 + 40 = {0}", result);

Func<int, int, string> funcTarget2 = new Func<int, int, string>(SumToString);
string sum = funcTarget2(90, 300);
Console.WriteLine(sum);
```

鉴于Action<>和Func<>节省了手工构建自定义委托的步骤，你可能想知道是否应该总是使用它们。和许多其他编程问题一样，答案是“视情况而定”。在很多情况下Action<>和Func<>都是首选。但如果你觉得一个具有自定义名称的委托更有助于捕获问题范畴，那么构建自定义委托不过就是一行代码的事儿。在本书剩余部分你将会看到这两种方式。

说明　很多重要的.NET API大量使用了Action<>和Func<>委托，包括并行程序框架和LINQ等。

我们对于.NET委托类型的初步介绍就到此为止。在本章小结和第19章介绍多线程和异步调用时还会涉及委托的一些其他细节。下面，我们来看看C# event关键字。

源代码　ActionAndFuncDelegates项目的源代码位于Chapter 10子目录下。

10.5 C#事件

委托确实是一个有趣的结构，它允许内存中的对象进行双向对话。然而，你可能会同意，从头使用委托会有一些重复代码（定义委托，声明必要的成员变量以及创建自定义的注册/注销方法来保护封装等）。

除了时间之外，这样使用委托来作为应用程序的回调机制会有另一个问题：如果我们没有把委托成员变量定义为私有的，调用者就可以直接访问委托对象。这样，调用者就可以把变量重新赋值为新的委托对象（实际上，也就删除了当前要调用的方法列表），更糟糕的是，调用者可以直接调用委托的调用列表。为了说明这个问题，考虑对之前CarDelegate实例的改写（和简化）：

```
public class Car
{
  public delegate void CarEngineHandler(string msgForCaller);

  // 这是一个公共成员
  public CarEngineHandler listOfHandlers;

  // 触发分解的通知
  public void Accelerate(int delta)
  {
    if (listOfHandlers != null)
      listOfHandlers("Sorry, this car is dead...");
  }
}
```

注意，我们不再有使用自定义的注册方法封装的私有委托成员变量。因为这些成员确实是公共的，调用者可以直接访问listOfHandlers成员变量，把这个类型重新分配给新的CarEngineHandler对象并且随时调用委托：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Agh!  No Encapsulation! *****\n");
    // 创建一个Car
    Car myCar = new Car();
    // 我们可以直接访问委托
    myCar.listOfHandlers = new Car.CarEngineHandler(CallWhenExploded);
    myCar.Accelerate(10);

    // 现在可以赋值一个全新的对象
    myCar.listOfHandlers = new Car.CarEngineHandler(CallHereToo);
    myCar.Accelerate(10);

    // 调用者还可以直接调用委托
    myCar.listOfHandlers.Invoke("hee, hee, hee...");
    Console.ReadLine();
  }

  static void CallWhenExploded(string msg)
  { Console.WriteLine(msg); }

  static void CallHereToo(string msg)
```

10

```
    { Console.WriteLine(msg); }
  }
```

公共的委托成员打破了封装，不仅会导致代码难以维护和调试，还会导致应用程序的安全风险！当前的示例的输出结果如下所示：

```
***** Agh!  No Encapsulation! *****

Sorry, this car is dead...
Sorry, this car is dead...
hee, hee, hee...
```

显然，我们不希望给其他应用程序改变委托指向的权力以及没有我们的许可直接调用成员的权力。

源代码　PublicDelegateProblem项目的源代码位于Chapter 10子目录下。

10.5.1　event关键字

为了简化自定义方法的构建来为委托调用列表增加和删除方法，C#提供了event关键字。在编译器处理event关键字的时候，它会自动提供注册和注销方法以及任何必要的委托类型成员变量。这些委托成员变量总是声明为私有的，因此不能直接从触发事件的对象访问它们。可以肯定的是，event关键字就像一块语法糖，只是节省了我们打字的时间。

定义一个事件分为两个步骤。首先，我们需要定义一个委托类型，它包含在事件触发时将要调用的方法。其次，通过C# event关键字用相关委托声明这个事件。

为了演示event关键字，创建一个新的控制台应用程序CarEvents。在Car类的迭代中会定义AboutToBlow和Exploded这两个事件。事件相关联的委托类型会被命名为CarEngineHandler。下面是对Car类的第一次修改：

```
public class Car
{
  // 这个委托用来与Car的事件协作
  public delegate void CarEngineHandler(string msg);

  // 这种汽车可以发送这些事件
  public event CarEngineHandler Exploded;
  public event CarEngineHandler AboutToBlow;
 ...
}
```

向调用者发送一个事件，就如通过名称和相关联委托定义的必需参数来指定事件这么简单。为确保调用者注册事件，需要在调用委托的方法之前检查这个事件是否是无效值。了解了这些之后，下面来看修改后的Car的Accelerate()方法：

```
public void Accelerate(int delta)
{
  // 如果car无法使用了，触发Exploded事件
  if (carIsDead)
  {
```

```
      if (Exploded != null)
        Exploded("Sorry, this car is dead...");
    }
    else
    {
      CurrentSpeed += delta;

      // 已经无法使用了吗
      if (10 == MaxSpeed - CurrentSpeed
        && AboutToBlow != null)
      {
        AboutToBlow("Careful buddy!  Gonna blow!");
      }

      // 还好着呢
      if (CurrentSpeed >= MaxSpeed)
        carIsDead = true;
      else
        Console.WriteLine("CurrentSpeed = {0}", CurrentSpeed);
    }
  }
```

这样，我们已经设定了Car对象发送两个自定义事件，这不再需要定义自定义注册函数，也不需要声明委托成员变量。稍后将看到这种新汽车的使用，但在此之前，让我们更深入考查一下事件的架构。

10.5.2 揭开事件的神秘面纱

C#事件事实上会扩展为两个隐藏的公共方法，一个带add_前缀，另一个带remove_前缀。前缀后面是C#事件的名称。例如，Exploded事件产生两个隐藏方法，名为add_Exploded()和remove_Exploded()。查看add_AboutToBlow()的CIL指令，将发现对Delegate.Combine()方法的调用。考虑下面的部分CIL代码：

```
.method public hidebysig specialname instance void
add_AboutToBlow(class CarEvents.Car/CarEngineHandler 'value') cil managed
{
...
  call class [mscorlib]System.Delegate
  [mscorlib]System.Delegate::Combine(
    class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
...
}
```

可以想到，remove_AboutToBlow()方法将间接调用Delegate.Remove()方法：

```
.method public hidebysig specialname instance void
  remove_AboutToBlow(class CarEvents.Car/CarEngineHandler 'value')
  cil managed
{
...
  call class [mscorlib]System.Delegate
    [mscorlib]System.Delegate::Remove(
      class [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
...
}
```

最后，代表事件本身的CIL代码将使用.addon和.removeon指令对应要调用的add_XXX()和remove_XXX()方法的名称：

10

```
.event CarEvents.Car/EngineHandler AboutToBlow
{
  .addon void CarEvents.Car::add_AboutToBlow
    (class CarEvents.Car/CarEngineHandler)

  .removeon void CarEvents.Car::remove_AboutToBlow
    (class CarEvents.Car/CarEngineHandler)
}
```

现在我们已经了解了如何构建一个能够发送C#事件的类，并知道事件只是为了节省键入时间，下一个大问题就是如何在调用者这边监听传入的事件。

10.5.3　监听传入的事件

C#事件也简化了注册调用者事件处理程序的操作。现在无需指定自定义辅助方法，调用者仅需使用+=和-=操作符即可（操作符将在后台触发正确的add_XXX()或remove_XXX()方法）。注册一个事件要遵循以下模式：

```
// NameOfObject.NameOfEvent += new RelatedDelegate(functionToCall);
//
Car.EngineHandler d = new Car.CarEventHandler(CarExplodedEventHandler)
myCar.Exploded += d;
```

要与事件源断开时，使用-=操作符：

```
// NameOfObject.NameOfEvent -= new RelatedDelegate(functionToCall);
//
myCar.Exploded -= d;
```

由于有这些固定的模式，我们现在使用C#的事件注册语法修改Main()方法，如下所示：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Events *****\n");
    Car c1 = new Car("SlugBug", 100, 10);

    // 注册事件处理程序
    c1.AboutToBlow += new Car.CarEngineHandler(CarIsAlmostDoomed);
    c1.AboutToBlow += new Car.CarEngineHandler(CarAboutToBlow);

    Car.CarEngineHandler d = new Car.CarEngineHandler(CarExploded);
    c1.Exploded += d;

    Console.WriteLine("***** Speeding up *****");
    for (int i = 0; i < 6; i++)
       c1.Accelerate(20);

    // 从调用列表中移除CarExploded方法
    c1.Exploded -= d;

    Console.WriteLine("\n***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);
    Console.ReadLine();
  }
```

```
    public static void CarAboutToBlow(string msg)
    { Console.WriteLine(msg); }

    public static void CarIsAlmostDoomed(string msg)
    { Console.WriteLine("=> Critical Message from Car: {0}", msg); }

    public static void CarExploded(string msg)
    { Console.WriteLine(msg); }
  }
```

为了进一步简化事件注册，可以使用方法组转换。考虑下面的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Events *****\n");
  Car c1 = new Car("SlugBug", 100, 10);

  // 注册事件处理程序
  c1.AboutToBlow += CarIsAlmostDoomed;
  c1.AboutToBlow += CarAboutToBlow;
  c1.Exploded += CarExploded;

  Console.WriteLine("***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);

  c1.Exploded -= CarExploded;

  Console.WriteLine("\n***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);

  Console.ReadLine();
}
```


10.5.4 使用Visual Studio简化事件注册

Visual Studio都提供了协助注册事件处理过程的机制。当我们在事件注册操作中应用+=操作符时，将看到一个IntelliSense窗口，提示我们按Tab键来自动完成相关联的委托实例（如图10-2所示），这一过程可使用方法组转换语法捕获。

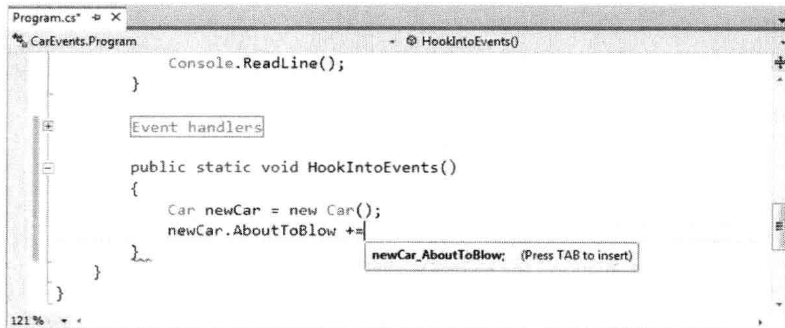


图 10-2 委托选择智能感知

按下Tab键后，IDE提示键入要创建的事件处理程序的名称（也可以接受默认的名称），如图10-3所示。

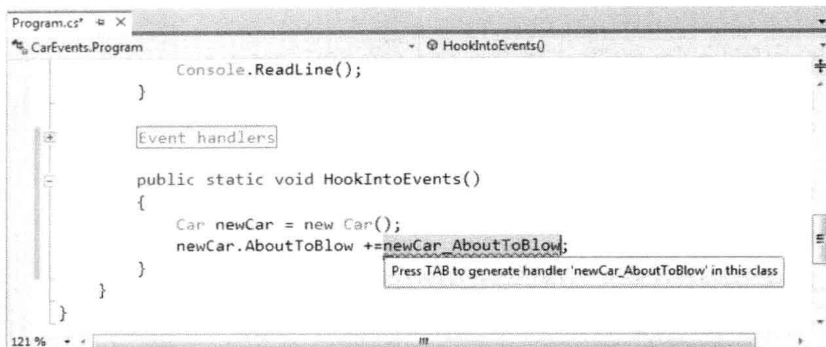


图10-3　委托目标对象格式智能感知

如果再次按Tab键，IDE将以委托对象的正确格式提供存根代码（注意，由于事件被注册在一个静态方法Main()里，所以这个方法已经被声明为静态的）：

```
static void newCar_AboutToBlow(string msg)
{
// 在这里添加你的代码
throw new NotImplementedException();
}
```

这种智能感知特性可用于基础类库中的所有.NET事件。这个IDE特性使我们不必再去搜索.NET帮助系统来断定某个事件要使用的正确委托和委托目标对象的正确格式，所以非常节省时间。

源代码　CarEvents项目的源代码位于Chapter 10子目录下。

10.5.5　创建自定义的事件参数

我们可以对Car类做最后一步改进，以符合微软推荐的事件模式。查看基础类库中某个类型发送的事件时，会发现底层委托的第一个参数是一个System.Object，第二个参数是一个派生自System.EventArgs的子类型。

System.Object参数表示一个对发送事件的对象（例如Car对象）的引用，第二个参数则表示与该事件相关的信息。System.EventArgs基类表示一个不发送任何自定义信息的事件：

```
public class EventArgs
{
  public static readonly EventArgs Empty;
  public EventArgs();
}
```

对于简单的事件来说，我们可以直接传递一个EventArgs的实例。但如果要传递自定义数据，应该构建一个派生自EventArgs的类。在这个示例中，假定我们有一个名为CarEventArgs的类，它保存一个字符串，表示要发送给接收者的信息：

```
public class CarEventArgs : EventArgs
{
  public readonly string msg;
  public CarEventArgs(string message)
  {
    msg = message;
  }
}
```

这样，我们就可以修改CarEventHandler委托了，如下所示（事件将保持不变）：

```
public class Car
{
  public delegate void CarEngineHandler(object sender, CarEventArgs e);
...
}
```

当在Accelerate()方法中触发我们的事件时，需要提供对当前Car对象的一个引用（通过this关键字）和CarEventArgs类型的一个实例。例如，考虑下面的部分更新：

```
public void Accelerate(int delta)
{
  // 如果Car无法使用了，触发Exploded事件
  if (carIsDead)
  {
    if (Exploded != null)
      Exploded(this, new CarEventArgs("Sorry, this car is dead..."));
  }
...
}
```

对于调用者，我们要做的就是修改事件处理程序来接收传入的参数，并通过只读字段获取消息。例如：

```
public static void CarAboutToBlow(object sender, CarEventArgs e)
{
  Console.WriteLine("{0} says: {1}", sender, e.msg);
}
```

如果接收者想与发送事件的对象交互，我们可以显式强制类型转换System.Object。这样，就可以使用传递给事件通知对象中的任何公共成员：

```
public static void CarAboutToBlow(object sender, CarEventArgs e)
{
  // 为安全起见，在强制类型转换前做一次运行时检查
  if (sender is Car)
  {
    Car c = (Car)sender;
    Console.WriteLine("Critical Message from {0}: {1}", c.PetName, e.msg);
  }
}
```

源代码 PrimAndProperCarEvents项目的源代码位于Chapter 10子目录下。

10.5.6 泛型EventHandler<T>委托

由于很多自定义委托接受object作为第一个参数，EventArgs派生类型作为第二个参数，我们可以

通过使用泛型EventHandler<T>类型来进一步简化之前的示例，其中T就是自定义的EventArgs类型。考虑如下对Car类型的更新（注意我们不再需要定义一个自定义委托类型）：

```
public class Car
{
  public event EventHandler<CarEventArgs> Exploded;
  public event EventHandler<CarEventArgs> AboutToBlow;
...
}
```

Main()方法现在就可以在之前定义CarEventHandler的任何地方使用EventHandler<CarEventArgs>（这次又使用了方法组转换）：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Prim and Proper Events *****\n");

  // 像平常一样创建一个Car
  Car c1 = new Car("SlugBug", 100, 10);

  // 注册事件处理程序
  c1.AboutToBlow += CarIsAlmostDoomed;
  c1.AboutToBlow += CarAboutToBlow;

  EventHandler<CarEventArgs> d = new EventHandler<CarEventArgs>(CarExploded);
  c1.Exploded += d;
...
}
```

太好了！我们已经学习了C#委托和事件的核心部分。这些信息可以满足所有回调需求，不过在本章最后我们还是来看一些简化方式，特别是匿名方法和Lambda表达式。

源代码　PrimAndProperCarEvents（泛型）项目的源代码位于Chapter 10子目录下。

10.6　C#匿名方法

可以看到，当一个调用者想监听传进来的事件时，它必须定义一个唯一的与相关联委托签名匹配的方法，例如：

```
class Program
{
  static void Main(string[] args)
  {
    SomeType t = new SomeType();

    // 假定"SomeDelegate"指向不带参数且无返回值的方法
    t.SomeEvent += new SomeDelegate(MyEventHandler);
  }

  // 一般情况下，它仅被SomeDelegate对象调用
  public static void MyEventHandler()
  {
    // 事件触发时执行某些操作
  }
}
```

思考一下就会发现，MyEventHandler()这样的方法很少会被调用委托之外的任何程序所调用。从生产效率的角度来说，手工定义一个由委托对象调用的方法显得有点烦琐，不会很受欢迎。

为了解决这一情况，现在可以在事件注册时直接将一个委托与一段代码相关联。这种代码的正式名称为匿名方法。为了说明它的基本语法，查看下面的Main()方法，它使用匿名方法处理Car类发送的事件，而不是使用命名的事件处理程序：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Anonymous Methods *****\n");
    Car c1 = new Car("SlugBug", 100, 10);

    // 注册事件处理程序作为匿名方法
    c1.AboutToBlow += delegate
    {
      Console.WriteLine("Eek! Going too fast!");
    };

    c1.AboutToBlow += delegate(object sender, CarEventArgs e)
    {
      Console.WriteLine("Message from Car: {0}", e.msg);
    };

    c1.Exploded += delegate(object sender, CarEventArgs e)
    {
      Console.WriteLine("Fatal Message from Car: {0}", e.msg);
    };

    // 这最终会触发事件
    for (int i = 0; i < 6; i++)
      c1.Accelerate(20);

    Console.ReadLine();
  }
}
```

10

说明 匿名方法中最后一个大括号必须以分号结束，否则，将产生一个编译错误。

请注意，Program类型不用再定义特定的静态事件处理程序，如CarAboutToBlow()或是CarExploded()了。未命名（即匿名）方法将在调用者使用+=语法处理事件时被内联定义。匿名方法的基本语法符合下面的伪代码：

```
class Program
{
  static void Main(string[] args)
  {
    SomeType t = new SomeType();
    t.SomeEvent += delegate (optionallySpecifiedDelegateArgs)
    { /* statements */ };
  }
}
```

在上一个Main()方法中处理第一个AboutToBlow事件时，注意没有定义委托传递的参数：

```
c1.AboutToBlow += delegate
{
  Console.WriteLine("Eek! Going too fast!");
};
```

严格来讲，我们不需要接收由指定事件发送的传入参数。但如果想使用可能传入的参数，需要通过委托类型指定参数原型（就像AboutToBlow和Exploded事件的第二个处理程序中一样），例如：

```
c1.AboutToBlow += delegate(object sender, CarEventArgs e)
{
  Console.WriteLine("Critical Message from Car: {0}", e.msg);
};
```

访问本地变量

匿名方法非常有趣，它使我们能访问定义它们的方法（称为定义方法）的本地变量。这些变量称为匿名方法的外部变量。有关匿名方法作用域与定义方法的作用域之间的交互，有几个重要的知识点，如下所示。

- 匿名方法不能访问定义方法中的ref或out参数。
- 匿名方法中的本地变量不能与外部方法中的本地变量重名。
- 匿名方法可以访问外部类作用域中的实例变量（或静态变量）。
- 匿名方法内的本地变量可以与外部类的成员变量同名（本地变量的作用域不同，可以隐藏外部类的成员变量）。

假定Main()方法定义了一个局部的整数变量aboutToBlowCounter。在处理AboutToBlow事件的匿名方法内部，计数器每次增加1并在Main()结束前输出点数：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Anonymous Methods *****\n");
  int aboutToBlowCounter = 0;

  // 像平常一样生成一个Car
  Car c1 = new Car("SlugBug", 100, 10);

  // 注册事件处理程序作为匿名方法
  c1.AboutToBlow += delegate
  {
    aboutToBlowCounter++;
    Console.WriteLine("Eek!  Going too fast!");
  };

  c1.AboutToBlow += (object sender, CarEventArgs e)
  {
    aboutToBlowCounter++;
    Console.WriteLine("Critical Message from Car: {0}", e.msg);
  };

...

  Console.WriteLine("AboutToBlow event was fired {0} times.",
    aboutToBlowCounter);
  Console.ReadLine();
}
```

如果读者运行了这段修改后的Main()方法，将发现最后Console.WriteLine()报告，AboutToBlow事件触发了两次。

源代码 AnonymousMethods项目的源代码位于Chapter 10子目录下。

10.7 Lambda 表达式

在讨论.NET事件架构的最后，我们来讲述C#的Lambda表达式。在本章前面已经解释过，C#支持内联处理事件，通过直接把一段代码语句赋值给事件（使用匿名方法），而不是构建被底层委托调用的独立方法。Lambda表达式只是用更简单的方式来写匿名方法，彻底简化了对.NET委托类型的使用。

为了对Lambda表达式的研究做准备，新建一个叫SimpleLambdaExpressions的控制台应用程序。现在，考虑泛型List<T>类的FindAll()方法。当你需要从一个集合中提取子集时，可以使用该方法，其原型如下：

```
// System.Collections.Generic.List<T>类中的方法
public List<T> FindAll(Predicate<T> match)
```

如你所见，该方法返回新的List<T>，表示数据子集。同时注意FindAll()方法的唯一参数是一个System.Predicate<T>类型的泛型委托。该委托指向任意以类型参数作为唯一输入参数并返回bool的方法：

```
// FindAll()方法使用该委托提取子集
public delegate bool Predicate<T>(T obj);
```

在调用FindAll()时，List<T>中的每一项都将传入Predicate<T>对象所指向的方法。方法在实现时将执行一些计算，来判断传入的数据是否符合标准，并返回true或false。如果返回true，该项将被添加到表示子集的新List<T>中（明白了吗）。

10

在学习Lambda表达式如何简化FindAll()的工作前，我们先直接使用委托对象这种普通方法。在Program类型中添加一个方法TraditionalDelegateSyntax()，它与System.Predicate<T>类型交互，找出整数List<T>中的偶数。

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Lambdas *****\n");
    TraditionalDelegateSyntax();

    Console.ReadLine();
  }

  static void TraditionalDelegateSyntax()
  {
    // 创建整数列表
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // 使用传统委托语法调用FindAll()
    Predicate<int> callback = new Predicate<int>(IsEvenNumber);
    List<int> evenNumbers = list.FindAll(callback);
```

```
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
      Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
  }

  // Predicate<>委托的目标
  static bool IsEvenNumber(int i)
  {
    // 这是一个偶数吗
    return (i % 2) == 0;
  }
}
```

在这里，方法IsEvenNumber()通过C#的取模操作符%来负责检查传入的整数参数是偶数还是奇数。如果我们执行应用程序，就会发现20、4、8和44被输出到了控制台上。

虽然这个使用委托的传统方式可以像预期那样工作，然而IsEvenNumber()方法只是在有限的环境中才会被调用。而且，如果调用FindAll()，就需要完整的方法定义。如果我们使用匿名方法来替代，代码就简洁多了。考虑如下Program类的新方法：

```
static void AnonymousMethodSyntax()
{
  // 建立整数列表
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // 现在使用匿名方法
  List<int> evenNumbers = list.FindAll(delegate(int i)
    { return (i % 2) == 0; } );

  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

在以上的代码中，我们并不是首先创建一个Predicate<T>委托类型，然后编写一个独立方法，而是使用了一个匿名方法。虽然这是正确的方向，但是仍然需要使用关键字delegate（或者一个强类型化的Predicate<T>），而且还需要保证输入参数百分之百匹配。我们认为，定义匿名方法的语法还是有点冗长。下面的代码可以更好地说明这个问题：

```
List<int> evenNumbers = list.FindAll(
  delegate(int i)
  {
    return (i % 2) == 0;
  }
);
```

我们可以使用Lambda表达式进一步简化对FindAll()方法的调用。使用新的语法时，底层的委托对象将会消失得无影无踪。请看下面的Program类的新方法：

```
static void LambdaExpressionSyntax()
```

```
{
  // 建立一个整数列表
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // 现在使用Lambda表达式
  List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

我们将奇怪的语句传递到方法`FindAll()`中，这些语句其实就是Lambda表达式。随着示例代码的更新，可以发现我们不再使用委托`Predicate<T>`（或者关键字`delegate`），而是用一个简短的Lambda表达式：`i => (i % 2) == 0`了。

在深入了解表达式的语法前，我们需要知道Lambda表达式可以应用于任何匿名方法或者强类型委托可以应用的场合，而且比匿名方法更节省编码时间。其实C#编译器只是把表达式翻译为使用委托`Predicate<T>`的标准匿名方法而已（可以使用ildasm.exe或reflector.exe进行验证），如下面的代码：

```
// Lambda表达式
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

被编译进下面的C#代码：

```
// ……变成了匿名方法
List<int> evenNumbers = list.FindAll(delegate (int i)
{
  return (i % 2) == 0;
});
```

10

10.7.1 剖析Lambda表达式

Lambda表达式是这样编写的：首先定义一个参数列表，“=>”标记（针对Lambda操作符的C#标记全部来自Lambda演算）紧随其后，然后就是处理这些参数的语句。从更高的级别讲，Lambda表达式可以理解为：

```
ArgumentsToProcess => StatementsToProcessThem
```

如果不使用`LambdaExpressionSyntax()`方法，则可以理解为：

```
// " i "就是我们的参数列表
// (i % 2) == 0就是处理"i"的表达式
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

Lambda表达式的参数既可以是显式类型化的，也可以是隐式类型化的。现在，表示参数`i`的数据类型（整型）是隐式类型化的。编译器可以根据整个Lambda表达式的上下文和底层委托推断出`i`是一个整型。尽管如此，我们也可以显式定义表达式的每一个参数的类型，如下用括号包围数据类型和变量名即可：

```
// 现在，我们显式定义参数的类型
List<int> evenNumbers = list.FindAll((int i) => (i % 2) == 0);
```

你可能已经发现，如果一个Lambda表达式拥有一个隐式类型化的参数，那么参数列表中的括号可以被省略。如果想以一致的方式编写Lambda表达式，那么你仍然可以使用括号：

```
List<int> evenNumbers = list.FindAll((i) => (i % 2) == 0);
```

最后，我们并没有使用括号包围表达式（使用了括号包围取模子表达式以确保它先于相等操作符执行）。Lambda表达式同样允许使用括号包围表达式，如下代码：

```
// 现在，我们使用括号包围表达式
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

至此你已经看到了编写Lambda表达式的多种方法，但是我们应该如何使用人类可识别的语言描述这个Lambda语句呢？请看（先把数学术语放一边）：

```
// 我的参数列表（一个整型i）将会被表达式(i % 2) == 0处理
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

10.7.2　使用多个语句处理参数

我们的第一个Lambda表达式是一条求出布尔类型的值的语句，但是我们知道很多委托目标需要执行多条代码语句。因此，C#允许使用一系列代码语句来定义Lambda表达式。当表达式必须使用多行代码处理参数时，你可以使用一对花括号确定这些语句的范围。请看下面对LambdaExpressionSyntax()方法进行更新的代码：

```
static void LambdaExpressionSyntax()
{
  // 创建整型列表
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // 现在使用语句块编写Lambda表达式
  List<int> evenNumbers = list.FindAll((i) =>
  {
    Console.WriteLine("value of i is currently: {0}", i);
    bool isEven = ((i % 2) == 0);
    return isEven;
  });

  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

现在，我们的参数列表（整型i）被一系列的代码语句处理。为了方便阅读，在调用Console.WriteLine()之后，我们把取模操作和结果返回分为两条语句。假设每个方法都在Main()中被调用：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Lambdas *****\n");
  TraditionalDelegateSyntax();
```

```
    AnonymousMethodSyntax();
    Console.WriteLine();
    LambdaExpressionSyntax();
    Console.ReadLine();
  }
```

程序输出结果如下所示：

```
***** Fun with Lambdas *****

Here are your even numbers:
20      4       8       44
Here are your even numbers:
20      4       8       44

value of i is currently: 20
value of i is currently: 1
value of i is currently: 4
value of i is currently: 8
value of i is currently: 9
value of i is currently: 44
Here are your even numbers:
20      4       8       44
```

源代码 SimpleLambdaExpressions项目的源代码位于Chapter 10子目录下。

10.7.3 含有多个（或零个）参数的Lambda表达式

到目前为止，我们编写的Lambda表达式都只含有一个参数。其实，Lambda表达式可以处理多个参数或者不提供任何参数。为了把问题说明白，创建一个新的控制台应用程序LambdaExpressions-MultipleParams。接下来，假设SimpleMath有如下更新：

```
public class SimpleMath
{
  public delegate void MathMessage(string msg, int result);
  private MathMessage mmDelegate;

  public void SetMathHandler(MathMessage target)
  {mmDelegate = target; }

  public void Add(int x, int y)
  {
    if (mmDelegate != null)
      mmDelegate.Invoke("Adding has completed!", x + y);
  }
}
```

我们可以看到，委托MathMessage需要两个参数。使用Lambda表达式的Main()如下所示：

```
static void Main(string[] args)
{
  // 使用Lambda表达式注册委托
```

```
    SimpleMath m = new SimpleMath();
    m.SetMathHandler((msg, result) =>
      {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});

    // 执行Lambda表达式
    m.Add(10, 10);
    Console.ReadLine();
  }
```

这里，为了简洁并没有使用强类型化，因此需要借助类型推断。但是我们可以像下面这样调用方法SetMathHandler()：

```
m.SetMathHandler((string msg, int result) =>
  {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});
```

最后，如果需要使用Lambda表达式与一个没有参数的委托交互，可以使用空括号表示表达式的参数列表。例如，假设我们定义了一个委托类型：

```
public delegate string VerySimpleDelegate();
```

你可以这样处理调用的结果：

```
// 在控制台输出"Enjoy your string!"
VerySimpleDelegate d = new VerySimpleDelegate( () => {return "Enjoy your string!";} );
Console.WriteLine(d.Invoke());
```

源代码　LambdaExpressionsMultipleParams项目的源代码位于Chapter 10子目录下。

10.7.4　使用Lambda表达式重新编写CarEvents示例

推荐使用Lambda表达式的原因是它为我们提供了一种简洁明了的方式来定义匿名方法（因此间接地简化了关于委托的编码工作）。现在让我们使用Lambda表达式重新编写第10章的PrimAndProperCarEvents项目。下面是项目的Program类的简化版本，它使用Lambda表达式语法（而不是传统的委托语法）挂接从Car对象发送的每个事件：

```
static void Main(string[] args)
{
  Console.WriteLine("***** More Fun with Lambdas *****\n");

  // 像平常一样创建一个Car对象
  Car c1 = new Car("SlugBug", 100, 10);

  // 使用Lambda表达式挂接事件
  c1.AboutToBlow += (sender, e) => { Console.WriteLine(e.msg);};
  c1.Exploded += (sender, e) => { Console.WriteLine(e.msg); };

  // 加速（这会触发事件）
  Console.WriteLine("\n***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.Accelerate(20);

  Console.ReadLine();
}
```

至此，我希望你可以了解Lambda表达式的整体角色和它是如何以“函数方式”与匿名方法和委托类型共同工作的。尽管你可能需要花一点时间来适应新的Lambda操作符(=>)，不过要始终记住Lambda表达式可以简化为如下简单的形式：

```
ArgumentsToProcess => StatementsToProcessThem
```

需要指出的是，LINQ编程模型使用了许多Lambda表达式来简化编码。第12章将会探讨LINQ。

源代码 CarEventsWithLambdas项目的源代码位于Chapter 10子目录下。

10.8 小结

本章讨论了许多可以让多个对象共同参与一个双向对话的方法。首先，我们讨论了C# `delegate`关键字，它用来间接构造一个派生自`System.MulticastDelegate`的类。可以看到委托对象保存着一个方法列表，可以在需要时调用这些方法。这些调用可能是同步进行的（使用`Invoke()`方法），也可能是异步的（通过`BeginInvoke()`方法和`EndInvoke()`方法）。还要说明的是，.NET委托类型的异步特性在第19章还会继续介绍。

接下来是C# `event`关键字，它与委托类型一起使用，可以简化发送事件通知到调用者的处理过程。通过转换成的CIL可以看出，.NET事件模型映射为对`System.Delegate`/`System.MulticastDelegate`类型的隐藏调用。从这个角度来说，C# `event`关键字纯粹是用以节省键入时间的可选方案。

随后，本章讨论了一个称为*匿名方法*的C#语言特性。使用这种语法构造，可以直接将一段代码与指定事件相关联。我们已经看到，匿名方法可以忽略事件发送的参数并能访问定义方法的外部变量。最后我们讨论了一种注册事件的简化方法——使用方法组转换。

最后，我们研究了C# Lambda操作符=>。这个语法是编写匿名方法的一个简化形式，可以把一堆参数传入语句组进行处理。.NET平台中接受委托对象参数的任意方法都可以用相关的Lambda表达式替换，并且通常都能够大大简化代码。

第 11 章 高级C#语言特性

本章将讨论许多高级的语法构造以加深大家对C#编程语言的理解。首先学习实现和使用*索引器方法*。这个C#机制允许我们构建能够以类似访问数组的语法来访问内部子类型的自定义类型。当学会如何构建索引器方法后，接下来讨论如何重载各种操作符（+、–、<、>等），并创建自定义显式类型转换与隐式类型转换，同时将介绍为什么要这样做。

接下来，我们将学习与LINQ相关的API中非常有用的话题（当然也可以在LINQ以外的上下文中使用它们）——特别是扩展方法和匿名类型。

最后，我们会学习如何创建“不安全”代码来直接操纵非托管指针。尽管C#应用程序很少使用指针，但在一些复杂的互操作场景中，理解如何使用指针还是非常有用的。

11.1 索引器方法

作为程序员，我们非常熟悉使用索引操作符（[]）访问包含在一个标准数组中的各个子项。例如：

```
static void Main(string[] args)
{
  // 使用索引操作符遍历传入的命令行参数
  for(int i = 0; i < args.Length; i++)
    Console.WriteLine("Args: {0}", args[i]);

  // 声明一个局部整数数组
  int[] myInts = { 10, 9, 100, 432, 9874};

  // 使用索引操作符访问每个元素
  for(int j = 0; j < myInts.Length; j++)
    Console.WriteLine("Index {0}  = {1} ", j,  myInts[j]);
  Console.ReadLine();
}
```

上面的代码绝不是什么新鲜事物。C#允许构建按照标准数组方式索引的自定义类和结构。顺理成章地能以这种方式访问子项的方法称为*索引器方法*（indexer method）。构建自定义集合类（泛型或非泛型）时，这个特殊的语言功能特别有用。

在探索如何实现一个自定义索引器之前，让我们来看一个运行中的例子。假定要在第9章（准确地说，是在IssuesWithNonGenericCollection项目）中开发的自定义类型PersonCollection里添加对索引器方法的支持。尽管还没添加索引器，但我们不妨先考虑新控制台应用程序SimpleIndexer的用法：

```
// 索引器允许我们以访问数组的方式访问各个子项
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Indexers *****\n");

    PersonCollection myPeople = new PersonCollection();

    // 使用索引器语法添加对象
    myPeople[0] = new Person("Homer", "Simpson", 40);
    myPeople[1] = new Person("Marge", "Simpson", 38);
    myPeople[2] = new Person("Lisa", "Simpson", 9);
    myPeople[3] = new Person("Bart", "Simpson", 7);
    myPeople[4] = new Person("Maggie", "Simpson", 2);

    // 现在使用索引器获取并显示每个子项
    for (int i = 0; i < myPeople.Count; i++)
    {
      Console.WriteLine("Person number: {0}", i);
      Console.WriteLine("Name: {0} {1}",
        myPeople[i].FirstName, myPeople[i].LastName);
      Console.WriteLine("Age: {0}", myPeople[i].Age);
      Console.WriteLine();
    }
  }
}
```

可见，索引器允许我们像操作一个标准数组那样操作内部子对象集合。现在大问题就来了：如何配置PersonCollection类（或其他任何自定义类/结构）使它支持这种功能呢？索引器看上去像是轻量级的修改过的C#属性。索引器最简单的创建形式是使用this[]语法。为此，相应地修改PersonCollection类：

```
// 给现有的类定义添加索引器
public class PersonCollection : IEnumerable
{
  private ArrayList arPeople = new ArrayList();

  // 类的自定义索引器
  public Person this[int index]
  {
    get { return (Person)arPeople[index]; }
    set { arPeople.Insert(index, value); }
  }
...
}
```

除了使用this关键字以外，索引器看上去和任何其他C#属性声明很相似。例如，get作用域会把当前对象返回给调用者。在这里，我们其实是使用ArrayList对象的索引器来完成的，就像该类支持索引器一样。set作用域负责添加新的Person对象，这是通过调用ArrayList的Insert()方法来完成的。

索引器是另一种语法便利手段，因为这种功能也可以使用普通的公共方法（如AddPerson()或GetPerson()）。不过，在自定义集合类型中支持索引器方法后，能很好地和.NET基础类库融为一体。

虽然在构建自定义集合时，构建索引器方法很常见，但是需要记住，泛型类型直接支持这个功能。考虑如下方法，它使用了Person对象的泛型List<T>。注意，我们现在就可以直接使用List<T>的索引器了，例如：

```
static void UseGenericListOfPeople()
{
  List<Person> myPeople = new List<Person>();
  myPeople.Add(new Person("Lisa", "Simpson", 9));
  myPeople.Add(new Person("Bart", "Simpson", 7));

  // 改变第一个人的索引器
  myPeople[0] = new Person("Maggie", "Simpson", 2);

  // 通过索引器获取和显示每一项
  for (int i = 0; i < myPeople.Count; i++)
  {
    Console.WriteLine("Person number: {0}", i);
    Console.WriteLine("Name: {0} {1}", myPeople[i].FirstName,
      myPeople[i].LastName);
    Console.WriteLine("Age: {0}", myPeople[i].Age);
    Console.WriteLine();
  }
}
```

源代码　SimpleIndexer项目的源代码位于Chapter 11子目录下。

11.1.1　使用字符串值索引对象

现在的PersonCollection类定义了一个允许调用者使用数值识别子项的索引器。不过要知道索引器方法，不一定非要这样做。假定我们更想把Person对象放到一个System.Collections.Generic.Dictionary<TKey, TValue>而不是ArrayList中。由于Dictionary类型允许使用字符串标记（例如一个人名）来访问其中包含的类型，我们可以设定新的索引器如下：

```
public class PersonCollection : IEnumerable
{
  private Dictionary<string, Person> listPeople =
    new Dictionary<string, Person>();

  // 这个索引器基于一个字符串索引返回一个Person
  public Person this[string name]
  {
    get { return (Person)listPeople[name]; }
    set { listPeople[name] = value; }
  }

  public void ClearPeople()
  { listPeople.Clear(); }

  public int Count
  { get { return listPeople.Count; } }

  IEnumerator IEnumerable.GetEnumerator()
  { return listPeople.GetEnumerator(); }
}
```

调用者现在可以与包含的Person对象交互，如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Indexers *****\n");

  PersonCollection myPeople = new PersonCollection();

  myPeople["Homer"] = new Person("Homer", "Simpson", 40);
  myPeople["Marge"] = new Person("Marge", "Simpson", 38);

  // 获取"Homer"并输出数据
  Person homer = myPeople["Homer"];
  Console.WriteLine(homer.ToString());

  Console.ReadLine();
}
```

如果要直接使用泛型Dictionary<Tkey, TValue>类型，可以直接获得索引器方法功能，而不用构建一个自定义的支持字符串索引器的非泛型类。不过，一定要清楚的是，任何索引器的数据类型都取决于支持的集合类型允许调用者获取子项的方式。

源代码 StringIndexer项目的源代码位于Chapter 11子目录下。

11.1.2 重载索引器方法

索引器方法可以在单个类或结构上被重载。因此，如果要让调用者通过数字索引或字符串值访问子项，就可能需要为一个字符串类型定义多个索引器。例如，在ADO.NET中（.NET原生数据库访问API），可能会想起DataSet类支持一个叫Tables的属性，它返回的是强类型的DataTableCollection类型。结果是DataTableCollection定义了3个索引器来获取和设置DataTable对象，一个是根据顺序位置，其他两个是根据友好字符串名称和可选的包含命名空间，如下所示：

```
public sealed class DataTableCollection : InternalDataCollectionBase
{
...
  // 重载的索引器
  public DataTable this[string name] { get; }
  public DataTable this[string name, string tableNamespace] { get; }
  public DataTable this[int index] { get; }
}
```

基础类库中的类型支持索引器方法是非常普遍的。因此，即使当前的项目不一定需要为类和结构构建自定义索引器，但很多类型已经支持了这个语法。

11.1.3 多维的索引器

如果真想特立独行，也可以创建一个传入多个参数的索引器。假定有一个以二维数组方式存储子项的自定义集合，可以设定索引器方法如下：

```
public class SomeContainer
{
  private int[,] my2DintArray = new int[10, 10];
```

```
  public int this[int row, int column]
  {  /* 从二维数组中取值或赋值*/  }
}
```

除非要构建高度程式化的自定义集合类，否则一般不会需要构建多维索引器。尽管如此，ADO.NET再次展示了这种构造的用途。ADO.NET中的DataTable实际上是行和列的集合，很像是方格纸或微软Excel电子表格的一般结构。

DataTable对象通常用"数据适配器"填充，但下面的代码演示了如何手动在内存中创建包含3列（名、姓和年龄）的DataTable。注意，在向DataTable中添加一行时，我们使用多维索引器访问行的每一列（需要在代码文件中引入System.Data命名空间）。

```
static void MultiIndexerWithDataTable()
{
  // 创建一个包含3列的简单DataTable
  DataTable myTable = new DataTable();
  myTable.Columns.Add(new DataColumn("FirstName"));
  myTable.Columns.Add(new DataColumn("LastName"));
  myTable.Columns.Add(new DataColumn("Age"));

  // 向表中添加一行
  myTable.Rows.Add("Mel", "Appleby", 60);

  // 使用多维索引器获取第一行中的详细内容
  Console.WriteLine("First Name: {0}", myTable.Rows[0][0]);
  Console.WriteLine("Last Name: {0}", myTable.Rows[0][1]);
  Console.WriteLine("Age : {0}", myTable.Rows[0][2]);
}
```

我们将在第21章中详细介绍ADO.NET，因此不要担心以上那些不熟悉的代码。该示例主要用来说明索引器方法支持多维，并且如果使用正确，可以简化与自定义集合中子对象的交互。

11.1.4　在接口类型上定义索引器

最后，要知道索引器可以在指定.NET接口上定义，这样实现类型就可以提供自定义实现。

以下是一个接口的简单示例，它定义了使用数字索引器获取字符串对象的协议：

```
public interface IStringContainer
{
  string this[int index] { get; set; }
}
```

任何实现了该接口的类和结构都必须支持一个可读写的索引器，它使用数字值操作子项。下面是一个这种类的部分实现：

```
class SomeClass : IStringContainer
{
private List<string> myStrings = new List<string>();

  public string this[int index]
  {
    get { return myStrings[index]; }
    set { myStrings.Insert(index, value); }
  }
}
```

本章的第一个话题到这里就结束了。现在我们来看下一个语言特性，它可以使你构建的自定义类或结构响应C#的内置操作符。下面请允许我介绍操作符重载的概念。

11.2 操作符重载

和其他编程语言一样，C#有一组用来完成内建类型基本操作的操作符。例如，我们知道+操作符可以用于两个整数以得出一个更大的整数：

```
// 整数间的+操作符
int a = 100;
int b = 240;
int c = a + b;  // c现在是340
```

这不是什么新鲜事物，但你是否注意过同样的+操作符可被用于大多数内建的C#数据类型？举个例子，思考以下代码：

```
// 字符串间的+操作符
string s1 = "Hello";
string s2 = " world!";
string s3 = s1 + s2;  // s3现在是"Hello world!"
```

+操作符本身可以根据所提供的数据类型（本例中是字符串和整数）不同而有不同的功能。当+操作符被用于数字类型时，结果就是操作数之和；而当+操作符用于字符串类型时，结果就是字符串的串联。

C#语言允许我们构建自定义类型和结构，它们也能对同一组基本操作符（如+）做出不同的反应。尽管并非所有的C#操作符都能重载，但大多数都是可以的，如表11-1所示。

表11-1 C#操作符的可重载性

<table>
<tr><th>C#操作符</th><th>可重载性</th></tr>
<tr><td>+, -, !, ~, ++, - -, true, false</td><td>这组一元操作符可被重载</td></tr>
<tr><td>+, -, *, /, %, &, |, ^, <<, >></td><td>这组二元操作符可被重载</td></tr>
<tr><td>= =, !=, <, >, <=, >=</td><td>比较操作符可被重载。C#要求配套的操作符（即<和>、<=和>=、==和!=）一起重载</td></tr>
<tr><td>[]</td><td>[]操作符不可重载。但本章前面看到，索引器构造提供了同样的功能</td></tr>
<tr><td>()</td><td>()操作符不可重载。在本章稍后将会看到，自定义转换方法提供了同样的功能</td></tr>
<tr><td>+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=</td><td>简写赋值操作符不可重载。但当重载相关的二元操作符时，它们也能随之具有相应的新功能</td></tr>
</table>

11.2.1 重载二元操作符

为说明重载二元操作符的过程，构建新的控制台应用程序OverloadedOps，设想其中定义了以下简单的Point类：

```
// 仅是一个简单的C#类
public class Point
```

```
{
  public int X {get; set;}
  public int Y {get; set;}

  public Point(int xPos, int yPos)
  {
    X = xPos;
    Y = yPos;
  }

  public override string ToString()
  {
    return string.Format("[{0}, {1}]", this.X, this.Y);
  }
}
```

从逻辑上讲，把Point加到一起是有意义的。例如，把两个Point变量加到一起，将会得到一个新的Point，它是X和Y值的和。同样，从一个Point中减去另一个也是有用的。理想情况下，可能写出如下代码：

```
// 加减两个Point
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Overloaded Operators *****\n");

  // 创建两个点
  Point ptOne = new Point(100, 100);
  Point ptTwo = new Point(40, 40);
  Console.WriteLine("ptOne = {0}", ptOne);
  Console.WriteLine("ptTwo = {0}", ptTwo);

  // 将两个点相加得到一个更大的点
  Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

  // 将两个点相减得到一个更小的点
  Console.WriteLine("ptOne - ptTwo: {0} ", ptOne - ptTwo);
  Console.ReadLine();
}
```

然而，对于现在的Point，我们会收到一个编译时错误，因为Point类型不知道如何处理+或-操作符。C#提供operator关键字来允许自定义类型对内建操作符作出不同的反应，operator关键字只可与static关键字联合使用。当我们重载一个二元操作符（例如+与-）时，将传入与指定类类型相同的两个参数，本例中为Point。代码如下：

```
// 更加智能的Point类型
public class Point
{
...
  // 重载+操作符
  public static Point operator + (Point p1, Point p2)
  {
    return new Point(p1.X + p2.X, p1.Y + p2.Y);
}

  // 重载-操作符
  public static Point operator - (Point p1, Point p2)
  {
    return new Point(p1.X - p2.X, p1.Y - p2.Y);
  }
}
```

+操作符背后的逻辑很简单，仅是返回一个基于传入Point参数字段分别相加的全新Point对象。当写pt1+pt2的时候，可以预想它将隐式调用静态操作符+方法，如下所示：

```
// 伪代码: Point p3 = Point.operator+ (p1, p2)
Point p3 = p1 + p2;
```

同理，p1 - p2映射如下所示：

```
// 伪代码: Point p4 = Point.operator- (p1, p2)
Point p4 = p1 - p2;
```

有了这样的更新，程序就可以编译了，现在我们会发现可以加或减Point对象，输出结果如下：

```
ptOne = [100, 100]
ptTwo = [40, 40]
ptOne + ptTwo: [140, 140]
ptOne - ptTwo: [60, 60]
```

如果我们正在重载二元操作符，就不需要传入两个相同类型的参数。如果这样做有意义，可以有一个参数不一样。例如，这里是重载后的+操作符，它允许调用者根据数值的差值来获取新的Point：

```
public class Point
{
...
  public static Point operator + (Point p1, int change)
  {
    return new Point(p1.X + change, p1.Y + change);
  }

  public static Point operator + (int change, Point p1)
  {
    return new Point(p1.X + change, p1.Y + change);
  }
}
```

注意，如果你希望以这两种顺序传递参数，就需要该方法的两个版本（你不能只定义其中一个方法，而希望编译器自动支持另一个）。我们现在就可以如下使用新版本的+操作符：

```
// 输出[110,110]
Point biggerPoint = ptOne + 10;
Console.WriteLine("ptOne + 10 = {0}", biggerPoint);

// 输出[120,120]
Console.WriteLine("10 + biggerPoint = {0}", 10 + biggerPoint);
Console.WriteLine();
```

11.2.2 +=与-=操作符

如果你是由C++背景转到C#来的，可能会因为无法重载简写赋值操作符（+=、-=等）而感到遗憾。其实在C#中，如果一个类型重载了相关的二元操作符，这些简写赋值操作符会自动具有相应的新功能。因此，因为Point结构已经重载了+和-操作符，可以编写如下代码：

```
// 重载二元操作符时，能产生新的简写赋值操作符
static void Main(string[] args)
```

```
{
...
  // 自动改变功能的 +=
  Point ptThree = new Point(90, 5);
  Console.WriteLine("ptThree = {0}", ptThree);
  Console.WriteLine("ptThree += ptTwo: {0}", ptThree += ptTwo);

  // 自动改变功能的 -=
  Point ptFour = new Point(0, 500);
  Console.WriteLine("ptFour = {0}", ptFour);
  Console.WriteLine("ptFour -= ptThree: {0}", ptFour -= ptThree);
  Console.ReadLine();
}
```

11.2.3　重载一元操作符

C#也允许我们重载各种一元操作符，例如++与--。在重载一元操作符时，也必须使用static关键字和operator关键字，在本例中我们只传入一个与指定类/结构相同类型的参数。例如，如要用下列重载操作符修改Point：

```
public class Point
{
...
  // 将传入的Point的X/Y值加1
  public static Point operator ++(Point p1)
  {
    return new Point(p1.X+1, p1.Y+1);
  }

  // 将传入的Point的X/Y值减1
  public static Point operator --(Point p1)
  {
    return new Point(p1.X-1, p1.Y-1);
  }
}
```

也可以按如下代码增减Point的X和Y值：

```
static void Main(string[] args)
{
...
  // 向Point应用++和-- 一元操作符
  Point ptFive = new Point(1, 1);
  Console.WriteLine("++ptFive = {0}", ++ptFive);  // [2, 2]
  Console.WriteLine("--ptFive = {0}", --ptFive);  // [1, 1]

  // 使用相同的操作符进行后递增和后递减
  Point ptSix = new Point(20, 20);
  Console.WriteLine("ptSix++ = {0}", ptSix++);  // [20, 20]
  Console.WriteLine("ptSix-- = {0}", ptSix--);  // [21, 21]
  Console.ReadLine();
}
```

注意在之前的代码示例中，我们以两种独特的形式应用了自定义++和--操作符。在C++中，可以独立重载前/后递增/递减操作符。然而在C#中不行，递增/递减的返回值会自动进行正确的处理（即对于重载的++操作符，在表达式中pt++的值就是未修改的对象的值，而++pt就在表达式使用前应用了新值。）

11.2.4 重载相等操作符

回想一下第6章，System.Object.Equals()可以重写，实现引用类型间基于值（而不是基于引用）的比较。如果选择重写Equals()方法和与之密切相关的System.Object.GetHashCode()方法，重载相等操作符（==和!=）的意义不大。为便于说明，下面是修改后的Point类型：

```
// Point的变体也重载了==和!=操作符
public class Point
{
...
  public override bool Equals(object o)
  {
    return o.ToString() == this.ToString();
  }

  public override int GetHashCode()
  {
    return this.ToString().GetHashCode();
  }

  // 现在，让我们来重载==和!=操作符
  public static bool operator ==(Point p1, Point p2)
  {
    return p1.Equals(p2);
  }

  public static bool operator !=(Point p1, Point p2)
  {
    return !p1.Equals(p2);
  }
}
```

注意，==与!=操作符的实现仅仅是调用重写的Equals()方法就完成了大部分工作。因此，我们可以按如下代码修改Point类：

```
// 利用重载的相等操作符
static void Main(string[] args)
{
...
  Console.WriteLine("ptOne == ptTwo : {0}", ptOne == ptTwo);
  Console.WriteLine("ptOne != ptTwo : {0}", ptOne != ptTwo);
  Console.ReadLine();
}
```

可以看到，在比较两个对象时，使用广为人知的==和!=操作符而不是调用Object.Equals()操作符是相当直观的。如果为一个指定的类重载了相等操作符，一定要注意C#在重载==操作符时必须同时重载!=操作符（如果忘记这样做，编译器会提示你）。

11.2.5 重载比较操作符

在第8章里，介绍了一个类如何实现IComparable接口，以便于比较两个相似对象之间的关系。除此之外，也可以为该类重载比较操作符（<、>、<=和>=）。和相等操作符一样，C#要求：如果要重载<，则也必须重载>。<=与>=同理。如果Point类型重载了这些比较操作符，对象用户现在就可以比较Point，如下所示：

```
// 使用重载的 < 与 > 操作符
static void Main(string[] args)
{
...
  Console.WriteLine("ptOne < ptTwo : {0}", ptOne < ptTwo);
  Console.WriteLine("ptOne > ptTwo : {0}", ptOne > ptTwo);
  Console.ReadLine();
}
```

如果已经实现了IComparable接口（最好实现其等价的泛型接口），那么重载比较操作符就很容易了。下面是修改后的类定义：

```
// Point也可使用比较操作符比较
public class Point : IComparable<Point>
{
...
  public int CompareTo(Point other)
  {
    if (this.X > other.X && this.Y > other.Y)
      return 1;
    if (this.X < other.X && this.Y < other.Y)
      return -1;
    else
      return 0;
  }

  public static bool operator <(Point p1, Point p2)
  { return (p1.CompareTo(p2) < 0); }

  public static bool operator >(Point p1, Point p2)
  { return (p1.CompareTo(p2) > 0); }

  public static bool operator <=(Point p1, Point p2)
  { return (p1.CompareTo(p2) <= 0); }

  public static bool operator >=(Point p1, Point p2)
  { return (p1.CompareTo(p2) >= 0); }
}
```

11.2.6 操作符重载的最后思考

可以看到，C#允许我们构建可对各种内建的操作符做出不同反应的类型。在修改所有的类使它们支持这些行为之前，必须确保要重载的操作符逻辑上符合日常生活中的意义。

举个例子，假设为MiniVan类重载了乘法操作符。两个MiniVan对象相乘有什么意义呢？意义不大。如下所示的对MiniVan对象的使用会令人感到困惑。

```
// 极不直观
MiniVan newVan = myVan * yourVan;
```

重载操作符通常仅在构建原子数据类型时才有用。文本、点、矩形、分数和六边形都是操作符重载的很好候选。人、经理、汽车、数据库连接和网页却不是。就经验来说，如果一个重载操作符会使用户更难于理解该类型的功能，那就别用它。要谨慎使用这个特性。

源代码 OverloadedOps项目的源代码位于Chapter 11子目录下。

11.3 自定义类型转换

现在讨论与操作符重载紧密相关的自定义类型转换。为给稍后的讨论做好铺垫，快速回顾一下数值数据及相关类类型之间显式和隐式转换的概念。

11.3.1 回顾：数值转换

对于内建数值类型（sbyte、int、float等），如果在较小容器中存储较大数值，就需要一个显式转换，因为这可能导致数据丢失。这实际上就是告诉编译器："别理我，我知道我在做什么。"反过来，如果试图将较小类型放到指定目标类型中时不会丢失数据，将自动进行隐式转换：

```
static void Main()
{
  int a = 123;
  long b = a;       // 从int到long的隐式转换
  int c = (int) b;  // 从long到int的显式转换
}
```

11.3.2 回顾：相关的类类型间的转换

在第6章中我们看到，类类型可通过传统的继承关系（"is-a"关系）关联起来。这时，C#转换过程允许我们向类层次结构中的上级或下级进行强制类型转换。例如，一个派生类总可以被隐式强制类型转换为基类类型。而如果要在派生变量中存储基类类型，必须执行显式强制类型，如下所示：

```
// 两个相关的类类型
class Base{}
class Derived : Base{}

class Program
{
  static void Main(string[] args)
  {
    // 派生类向基类的隐式强制类型转换
    Base myBaseType;
    myBaseType = new Derived();


    // 在派生类型中存储基类引用必须显式强制类型转换
    Derived myDerivedType = (Derived)myBaseType;
  }
}
```

这个显式强制类型转换之所以行之有效，应归功于Base类和Derived类之间存在传统的继承关系。但如有没有公共父类（除了System.Object）不同层次结构中的两个类需要转换，又该怎么办呢？由于它们不存在继承关系，典型的强制类型转换操作于事无补。

与此相关，思考一下值类型（结构）。假设有两个名为Square和Rectangle的.NET结构。由于结构不能使用传统继承（因为它们是密封的），所以不能直接强制类型转换这些看起来差不多的类型。

在结构中构建辅助方法（如Rectangle.ToSquare()）的同时，C#允许我们构建能使用户类型响

应()操作符的自定义强制类型转换例程。因此，如果正确设定了结构，可使用如下语法显式转换这些结构：

```
// 将矩形转换为正方形
Rectangle rect;
rect.Width = 3;
rect.Height = 10;
Square sq = (Square)rect;
```

11.3.3　创建自定义转换例程

首先创建一个新的控制台应用程序CustomConversions。C#提供两个关键字explicit和implicit，可用来控制发生转换时类型的响应方式。假定有以下结构定义：

```
public struct Rectangle
{
  public int Width {get; set;}
  public int Height {get; set;}

  public Rectangle(int w, int h) : this()
  {
    Width = w; Height = h;
  }

  public void Draw()
  {
    for (int i = 0; i < Height; i++)
    {
      for (int j = 0; j < Width; j++)
      {
        Console.Write("*");
      }
      Console.WriteLine();
    }
  }

  public override string ToString()
  {
    return string.Format("[Width = {0}; Height = {1}]",
      Width, Height);
  }
}
public struct Square
{
  public int Length {get; set;}

  public Square(int l) : this()
  {
    Length = l;
  }


  public void Draw()
  {
    for (int i = 0; i < Length; i++)
    {
      for (int j = 0; j < Length; j++)
```

```
      {
        Console.Write("*");
      }
      Console.WriteLine();
    }
  }

  public override string ToString()
  { return string.Format("[Length = {0}]", Length); }

  // 矩形可显式转换为正方形
  public static explicit operator Square(Rectangle r)
  {
    Square s = new Square();
    s.Length = r.Height;
    return s;
  }
}
```

说明 你会发现在Square和Rectangle构造函数中，我显示链接到了默认构造函数。原因是如果结构使用了自动属性（如本例所示），所有自定义构造函数都必须显式调用默认构造函数，来初始化私有的后台字段。是的，这是C#中非常怪异的规则，但毕竟本章所讲的 是“高级语言特性“。

注意，Square类型的这个版本定义了一个显式转换操作符。如同重载操作符过程一样，转换例程使用C# operator关键字（结合使用explicit或implicit关键字）而且必须定义为静态的。传入参数是要转换的实体，而操作符类型是转换后的实体。

在这种情况下，假定能从矩形的宽度得到一个正方形（所有边都是等长的）。这样也就可以自由转换一个Rectangle为Square：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Conversions *****\n");

  // 创建一个矩形
  Rectangle r = new Rectangle(15, 4);
  Console.WriteLine(r.ToString());
  r.Draw();

  Console.WriteLine();

  // 根据矩形的宽度将r转换为正方形
  Square s = (Square)r;
  Console.WriteLine(s.ToString());
  s.Draw();
  Console.ReadLine();
}
```

输出结果如下所示。

```
***** Fun with Conversions *****

[Width = 15; Height = 4]

***************
***************
***************
***************

[Length = 4]
****
****
****
****
```

在一个代码段内转换Rectangle为Square可能意义不大，假定有一个参数为Square的函数：

```
// 该方法需要一个Square类型
static void DrawSquare(Square sq)
{
  Console.WriteLine(sq.ToString());
  sq.Draw();
}
```

在Square类型上使用显式强制类型转换操作，可以安全传入Rectangle类型供处理，如下所示：

```
static void Main(string[] args)
{
...
  // 将Rectangle转为Square来调用方法
  Rectangle rect = new Rectangle(10, 5);
  DrawSquare((Square)rect);
  Console.ReadLine();
}
```

11.3.4　Square类型的其他显式转换

现在可以显式转换Rectangle为Square了，让我们讨论其他一些显式转换。由于正方形每对边都是对称的，提供一个显式转换例程，允许调用者由整型类型转为Square类型也是有意义的（当然，将有一个边长等于传入的整数）。如果要修改Square，使调用者可转换Square为int，又该怎么办呢？下面是调用逻辑：

```
static void Main(string[] args)
{
...
  // 转换int为Square
  Square sq2 = (Square)90;
  Console.WriteLine("sq2 = {0}", sq2);

  // 转换Square为int
  int side = (int)sq2;
  Console.WriteLine("Side length of sq2 = {0}", side);
  Console.ReadLine();
}
```

下面是修改后的Square类：

```
public struct Square
{
...
  public static explicit operator Square(int sideLength)
  {
    Square newSq = new Square();
    newSq.Length = sideLength;
    return newSq;
  }

  public static explicit operator int (Square s)
  {return s.Length;}
}
```

客观来讲，将Square转为int可能并不是最直观（或最有用）的操作。不过，这确实让我们发现关于自定义转换例程非常重要的一点：只要代码语法书写正确，编译器并不关心由什么转换成什么。

同使用重载操作符一样，仅是因为可以为指定类型创建一个显式强制类型转换操作，并不意味着应该这样做。由于结构不能参与可以自由强制类型转换的传统继承，这种技术在大家创建.NET结构时很有用。

11.3.5 定义隐式转换例程

到现在为止，我们已经创建了各种显式转换操作。但是，下列隐式转换操作又会如何呢？

```
static void Main(string[] args)
{
...
  Square s3 = new Square();
  s3.Length = 83;

// 试图创建一个隐式强制类型转换吗
  Rectangle rect2 = s3;

  Console.ReadLine();
}
```

这段代码无法编译，因为没有为Rectangle类型提供隐式转换例程。如果返回类型或参数都相同，在同一类型中定义显式和隐式转换函数是非法的，这就是关键所在。这看起来好像是一个限制。不过，第二个关键点在于，当一个类型定义了隐式转换例程后，调用者使用显式强制类型转换语法是合法的！

困惑了吧？为清楚起见，我们使用C# implicit关键字给Rectangle结构添加一个隐式转换例程（注意，下列代码假设Rectangle的长度是由Square的长度乘2计算出来的）：

```
public struct Rectangle
{
...
  public static implicit operator Rectangle(Square s)
  {
    Rectangle r = new Rectangle();
    r.Height = s.Length;

    // 设定新矩形的长度为正方形长度乘2
    r.Width = s.Length * 2;
    return r;
  }
}
```

11

这次更新后，就可以按如下代码转换两种类型了：

```
static void Main(string[] args)
{
...
  // 隐式强制类型转换成功
  Square s3 = new Square();
  s3.Length= 7;
  Rectangle rect2 = s3;
  Console.WriteLine("rect2 = {0}", rect2);

  // 显式强制类型转换语法也成功
  Square s4 = new Square();
  s4.Length = 3;
  Rectangle rect3 = (Rectangle)s4;
  Console.WriteLine("rect3 = {0}", rect3);
  Console.ReadLine();
}
```

我们将结束关于定义自定义转换例程的讨论。关于重载操作符，要牢记这些语法仅是普通成员函数的简写表示法，从这个角度来说它们总是可选的。但只要使用正确，可以更自然地使用自定义结构，因为可以将它们当成是与继承相关联的真正的类类型。

源代码　CustomConversions项目的源代码位于Chapter 11子目录下。

11.4　扩展方法

.NET 3.5引入了*扩展方法*（extension method）的概念，它允许你在不直接修改原始类型的情况下，为类或结构添加新的方法或属性。那么在哪些情况下可以用到这个特性呢？考虑如下几种可能。

首先，假设我们给定了一个产品中的类。随着时间的推移，我们越来越清晰地发现该类应该支持一些新的成员。如果直接修改当前的类定义，将可能会破坏向后兼容性。使用该类的旧代码可能无法通过编译。一种确保向后兼容性的方式是创建集成现有父类的新派生类。但这样一来就有两个类需要维护。众所周知，代码维护是软件工程师的工作描述中最枯燥无聊的部分了。

现在考虑下面的情形。假设有一个结构（或封闭类），我们希望向其添加新的成员以在系统中展现其多态性。由于结构和封闭类不能扩展，我们只能再一次面对向后兼容的风险，向类型中添加新成员。

使用扩展方法，可以在不创建子类和直接修改类型的情况下修改类型。当然，该技术本质上是镜像原理。如果要在当前项目中使用扩展方法，也只能将这个新功能应用于类型。

11.4.1　定义扩展方法

当定义扩展方法时，你遇到的第1个限制就是必须把方法定义在静态类（参阅第5章）中，因此每一个扩展方法也必须声明为静态的。第2个限制就是所有的扩展方法都需要使用this关键字对第一个参数（并且仅对第一个参数）进行修饰。用this限定的参数表示被扩展的项。

为了演示，我们创建一个新的控制台应用程序，命名为ExtensionMethods。假设你正在编写一个工具类MyExtensions，并在该类中定义了两个扩展方法。第1个扩展方法允许.NET基础类库中的所有对象都将使用全新的方法DisplayDefiningAssembly()，该方法将使用命名空间System.Reflection中的类型来显示包含指定类型的程序集的名称。

说明 我们将在第15章中正式介绍反射API。如果你对其不甚了解，可以简单地理解为在运行时发现程序集、类型及类型成员的一种方式。

第2个扩展方法是ReverseDigits()，允许所有int将自己的值倒置。例如，整型1234调用ReverseDigits()，返回的结果是4321。考虑下面的类实现（确保导入了System. Reflection命名空间）：

```
static class MyExtensions
{
  // 本方法允许任何对象显示它所处的程序集
  public static void DisplayDefiningAssembly(this object obj)
  {
    Console.WriteLine("{0} lives here: => {1}\n", obj.GetType().Name,
      Assembly.GetAssembly(obj.GetType()).GetName().Name);
  }

  // 本方法允许任何整型返回倒置的副本，例如56将返回65
  public static int ReverseDigits(this int i)
  {
    // 把int翻译为string然后获取所有字符
    char[] digits = i.ToString().ToCharArray();

    // 现在反转数组中的项
    Array.Reverse(digits);

    // 放回string
    string newDigits = new string(digits);

    // 最后以int返回修改后的字符串
    return int.Parse(newDigits);
  }
}
```

注意每个扩展方法的第1个参数在定义参数类型前都使用了关键字this。大多数情况下，扩展方法的第一个参数表示被扩展的类型。DisplayDefiningAssembly()被定义为用于扩展System.Object，现在所有程序集中的类型都拥有这一新成员，因为Object是.NET平台所有类型的父类。尽管如此，ReverseDigits()只被定义为用于扩展整型类型，因此任何非整型对象尝试调用该方法都会产生编译错误。

说明 还需要知道的是，扩展方法可以拥有多个参数，但只有第1个参数可以使用关键字this进行修饰。其他参数将被视为供方法使用的普通传入参数。

11.4.2 在实例层次上调用扩展方法

现在我们已经定义了扩展方法，下面看看下面的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Extension Methods *****\n");

  // 本整型表示一个新的身份标识
  int myInt = 12345678;
  myInt.DisplayDefiningAssembly();

  // 下面是DataSet
  System.Data.DataSet d = new System.Data.DataSet();
  d.DisplayDefiningAssembly();

  // 下面是SoundPlayer
  System.Media.SoundPlayer sp = new System.Media.SoundPlayer();
  sp.DisplayDefiningAssembly();

  // 使用整型的新功能
  Console.WriteLine("Value of myInt: {0}", myInt);
  Console.WriteLine("Reversed digits of myInt: {0}", myInt.ReverseDigits());

  Console.ReadLine();
}
```

输出结果如下所示（去除了无关紧要的项）：

```
***** Fun with Extension Methods *****

Int32 lives here: => mscorlib

DataSet lives here: => System.Data

SoundPlayer lives here: => System

Value of myInt: 12345678
Reversed digits of myInt: 87654321
```

11.4.3　导入扩展方法

在定义包含扩展方法的类时，毫无疑问应该将其定义在.NET命名空间中。如果该命名空间与使用扩展方法的命名空间不同，就需要使用C#的using关键字。这样一来，你的代码文件就可以访问被扩展类型的所有扩展方法。需要牢记的是，如果没有显式导入正确的命名空间，扩展方法对当前C#代码文件不可用。

虽然表面上扩展方法是全局的，但其实它们受制于所处的命名空间。因此，如果我们把静态类MyExtensions封装在在名为MyExtensionMethods的命名空间中：

```
namespace MyExtensionMethods
{
  static class MyExtensions
  {
    ...
  }
}
```

这个项目中的其他命名空间都需要显式引入MyExtensionMethods命名空间以获取这些类中定义的扩展方法。

说明 不仅将扩展方法隔离在专门的.NET命名空间，还将它们放到专门的类库中，这种操作是很常见的。这样一来，新应用程序可以“选择”通过显示引用正确的库来进行扩展。第14章将详细介绍构建并使用自定义的.NET类库。

11.4.4 扩展方法的智能感知

由于扩展方法没有真正定义在要扩展的类型之中，因此在检查既有代码的时候肯定会很困惑。例如，假如导入了一个命名空间，一个团队成员在其中定义了许多扩展方法。我们自己在编码的时候可能会创建扩展类型的变量，使用点操作符，然后会找到不在原始类定义中的许多新方法！

可喜的是，Visual Studio的智能感知机制用独有的“向下”箭头图标标注所有扩展方法（如图11-1所示），在我们的屏幕上会是蓝色。

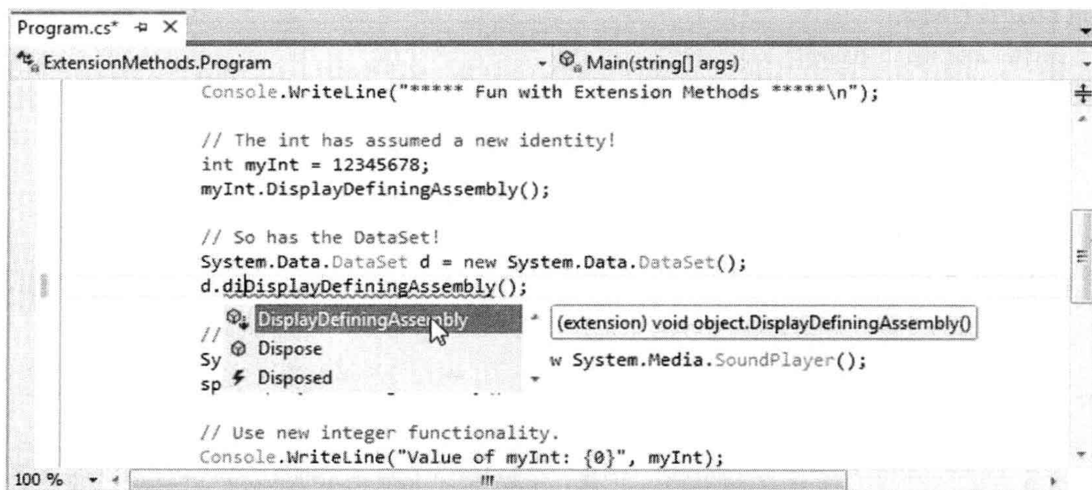


图11-1 扩展方法的智能感知

任何标记了这个图标的方法都表明这个方法不在原始类中定义而是通过扩展方法来定义。

源代码 ExtensionMethods项目的源代码位于Chapter 11子目录下。

11.4.5 扩展实现了指定接口的类型

至此，我们已经看到了如何使用扩展方法为类（结构也使用相同的语法）扩展新的功能。我们还可以定义扩展方法，让它只能扩展实现了正确接口的类或结构。例如，你可以达到“如果类或结构实现了IEnumerable<T>，即可得到新成员”的效果。当然，我们可以要求一个类型支持全部接口，包括我们自己定义的接口。

为了演示这一点，新建一个控制台应用程序InterfaceExtensions，我们的目的是为任何实现了IEnumerable的类型添加一个新方法，这些类型包括数组和很多非泛型集合类（第8章我们介绍过，泛型的IEnumerable<T>接口扩展了非泛型的IEnumerable接口）。在新项目中添加如下所示的扩展类：

```
static class AnnoyingExtensions
{
  public static void PrintDataAndBeep(this System.Collections.IEnumerable iterator)
  {
    foreach (var item in iterator)
    {
      Console.WriteLine(item);
      Console.Beep();
    }
  }
}
```

由于PrintDataAndBeep()方法可用于任何实现了IEnumerable的类或结构，我们可以通过下面的Main()方法进行测试：

```
static void Main( string[] args )
{
  Console.WriteLine("***** Extending Interface Compatible Types *****\n");

  // System.Array 实现 IEnumerable
  string[] data = { "Wow", "this", "is", "sort", "of", "annoying",
                    "but", "in", "a", "weird", "way", "fun!"};
  data.PrintDataAndBeep();

  Console.WriteLine();

  // List<T> 实现 IEnumerable
  List<int> myInts = new List<int>() {10, 15, 20};
  myInts.PrintDataAndBeep();

  Console.ReadLine();
}
```

这就结束了我们对C#扩展方法的研究。记住，在多态想扩展某个类型的功能，但却不想创建子类（或不能创建子类，比如类型被封闭）的时候，这个特殊语言特性会非常有用。另在本书后面将会看到，扩展方法在LINQ API中有着十分重要的作用。事实上，你将会看到在LINQ API中，最常见的被扩展项是实现了IEnumerable泛型版本的类或结构（相当出人意料）。

源代码 InterfaceExtensions项目的源代码位于Chapter 11子目录下。

11.5 匿名类型

作为一个面向对象的程序员，你应该知道定义类来表达要建模的给定项的状态和功能的好处。每当需要定义在项目间重用的类，同时这些类需要提供一系列方法、事件、属性和自定义构造函数时，我们通常创建一个C#类。

尽管如此，有时候你可能需要定义类来封装一些相关数据，但并不需要任何关联的方法、事件和

其他自定义的功能。此外，如果定义的类型只会被程序中少量方法使用呢？如果你明确知道类只会在少量的地方使用，那么像下面这样定义完整的类定义是非常麻烦的。为了强调这一点，以下是当需要创建一个“简单”的遵循典型基于值语义的数据类型时，可能需要编写的梗概。

```
class SomeClass
{
  // 定义一系列私有数据成员

  // 创建属性来封装每一个数据成员

  // 重写方法ToString()来输出关键数据成员

  // 重写GetHashCode()和Equals()来实现基于值的相等性
}
```

如你所见，这并不简单。你不但需要编写大量代码，而且系统中还多了一个类需要维护。对于这样的临时数据，动态地生成一个自定义数据类型将是非常有用的。例如，假设你需要构建一个接受若干传入参数的自定义方法。你会想在该方法作用域内使用这些参数来创建一个新的数据类型。此外，想要使用传统的ToString()方法快速打印数据，或者使用System.Object的其他成员。你可以使用匿名类型语法实现这些操作。

11.5.1 定义匿名类型

当定义一个匿名类型时，你需要使用新的关键字var（见第3章）和前面介绍的对象初始化语法（见第5章）。我们必须使用关键字var，因为编译器会在编译时自动生成新的类定义（我们永远无法在C#代码中看到该类的名字）。初始化语法将告诉编译器为新创建的类型创建私有的后台字段和（只读的）属性。

创建新的控制台应用程序，命名为AnonymousTypes。然后，在Program类中添加下面的方法，它将使用传入参数再动态地创建一个新的类型：

```
static void BuildAnonType( string make, string color, int currSp )
{
  //使用传入参数构建匿名类型
  var car = new { Make = make, Color = color, Speed = currSp };

  //注意，现在可以使用该类型获取属性数据
   Console.WriteLine("You have a {0} {1} going {2} MPH",
    car.Color, car.Make, car.Speed);

  // 匿名类型包含对System.Object中
  //每个虚方法的自定义实现，例如：
  Console.WriteLine("ToString() == {0}", car.ToString());
}
```

你可以在Main()中调用该方法。不过要注意的是，匿名类型也可以使用硬编码值创建，如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Anonymous Types *****\n");

  // 构建一个匿名对象表示一辆汽车
  var myCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };

  // 显示颜色并输出
```

```
  Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Make);

  // 调用辅助方法通过实参创建匿名类型
  BuildAnonType("BMW", "Black", 90);

  Console.ReadLine();
}
```

此时，我们只需要了解匿名类型允许我们以非常小的开销快速建立数据的“形状”。这项技术只是动态地创建新数据类型的一种方式，它支持通过属性来封装骨架，其行为符合基于值的语义。要理解最后这一点，我们来看看C#编译器如何在编译时构建匿名类型，以及如何重载System.Object的成员。

11.5.2　匿名类型的内部表示方式

所有的匿名类型都自动继承System.Object，因此它们都支持基类的每一个成员。我们可以在隐式类型化的myCar对象上调用方法ToString()、GetHashCode()、Equals()或者GetType()。假设我们的Program类已经定义了以下的静态帮助方法：

```
static void ReflectOverAnonymousType(object obj)
{
  Console.WriteLine("obj is an instance of: {0}", obj.GetType().Name);
  Console.WriteLine("Base class of {0} is {1}",
    obj.GetType().Name,
    obj.GetType().BaseType);
  Console.WriteLine("obj.ToString() == {0}", obj.ToString());
  Console.WriteLine("obj.GetHashCode() == {0}", obj.GetHashCode());
  Console.WriteLine();
}
```

现在我们在方法Main()中调用此方法，以myCar对象作为参数：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Anonymous Types *****\n");

  // 构建一个匿名类型表示一辆汽车
  var myCar = new {Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55};

  // 使用反射输出编译器生成的内容
  ReflectOverAnonymousType(myCar);
...

  Console.ReadLine();
}
```

程序的输出结果如下所示：

```
***** Fun with Anonymous Types *****

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() = { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() = -439083487
```

首先，注意在上面的例子中，myCar对象的类型是<>f__AnonymousType0`3（你的版本也许有所不同）。匿名类型的类型名完全由编译器决定，我们不能编写代码对此进行干涉。

最为重要的是，使用对象初始化语法定义的每一个名称/值对分别被映射为拥有相同名字的只读属性以及对应被该属性封装的私有数据成员。下面的C#代码大体上表示了编译器生成的myCar对象的类定义（同样可以使用ildasm.exe进行验证）：

```
internal sealed class <>f__AnonymousType0<<Color>j__TPar,
  <Make>j__TPar, <CurrentSpeed>j__TPar>
{
  // 只读字段
  private readonly <Color>j__TPar <Color>i__Field;
  private readonly <CurrentSpeed>j__TPar <CurrentSpeed>i__Field;
  private readonly <Make>j__TPar <Make>i__Field;

  // 默认构造函数
  public <>f__AnonymousType0(<Color>j__TPar Color,
     <Make>j__TPar Make, <CurrentSpeed>j__TPar CurrentSpeed);
  // 重写若干方法
  public override bool Equals(object value);
  public override int GetHashCode();
  public override string ToString();

  // 只读属性
  public <Color>j__TPar Color { get; }
  public <CurrentSpeed>j__TPar CurrentSpeed { get; }
  public <Make>j__TPar Make { get; }
}
```

11.5.3 方法ToString()和GetHashCode()的实现

我们可以发现匿名类型直接继承了System.Object，并且重写了方法Equals()、GetHashCode()和ToString()。其中ToString()根据每个名称/值对，生成并返回一个字符串：

```
public override string ToString()
{
  StringBuilder builder = new StringBuilder();
  builder.Append("{ Color = ");
  builder.Append(this.<Color>i__Field);
  builder.Append(", Make = ");
  builder.Append(this.<Make>i__Field);
  builder.Append(", CurrentSpeed = ");
  builder.Append(this.<CurrentSpeed>i__Field);
  builder.Append(" }");
  return builder.ToString();
}
```

GetHashCode()的实现使用每个匿名类型的成员变量计算出散列值作为System.Collections.Generic.EqualityComparer<T>类型的输入。使用GetHashCode()的实现，如果（也仅当）两个匿名类型有相同的属性并且被赋予了相同的值，就会产生相同的散列值。有了这个实现，匿名类型就完全可以包含在Hashtable容器中。

11.5.4　匿名类型的相等语义

我们可以看到，方法ToString()和GetHashCode()的重写实现相当简单，现在你可能想知道方法Equals()是如何被重写的。例如，如果你定义了两辆“匿名车辆”变量，它们定义了相同的名称/值对，那么两者是否相等呢？为了知道结果，首先需要更新Program类型：

```
static void EqualityTest()
{
  // 构建两个匿名类，拥有相同的名称/值对
  var firstCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };
  var secondCar = new { Color = "Bright Pink", Make = "Saab", CurrentSpeed = 55 };

  // 调用Equals()的结果是什么
  if (firstCar.Equals(secondCar))
    Console.WriteLine("Same anonymous object!");
  else
    Console.WriteLine("Not the same anonymous object!");

  // 使用==操作符进行比较的结果是什么
  if (firstCar == secondCar)
    Console.WriteLine("Same anonymous object!");
  else
    Console.WriteLine("Not the same anonymous object!");

  // 两个对象的类型是否相同
  if (firstCar.GetType().Name == secondCar.GetType().Name)
    Console.WriteLine("We are both the same type!");
  else
    Console.WriteLine("We are different types!");

  // 显示两个匿名类的细节
  Console.WriteLine();
  ReflectOverAnonymousType(firstCar);
  ReflectOverAnonymousType(secondCar);
}
```

假设从Main()中调用了EqualityTest()方法，输出结果将如下所示（多少有些令人吃惊）：

```
My car is a Bright Pink Saab.
You have a Black BMW going 90 MPH
ToString() == { Make = BMW, Color = Black, Speed = 90 }

Same anonymous object!
Not the same anonymous object!
We are both the same type!

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -439083487

obj is an instance of: <>f__AnonymousType0`3
Base class of <>f__AnonymousType0`3 is System.Object
obj.ToString() == { Color = Bright Pink, Make = Saab, CurrentSpeed = 55 }
obj.GetHashCode() == -439083487
```

当运行前面的测试代码时，你可以看到在第1个条件测试中，调用Equals()返回了true，因此屏幕输出“Same anonymous object!”。这是因为编译器重写的Equals()在判断对象相等时使用了基于值的语义（例如比较两个对象的每一个数据成员的值）。

尽管如此，第2个条件测试（使用C#的相等操作符==）输出了“Not the same anonymous object!”，这可能与部分人的答案有所不同。输出的结果之所以有点违反常理，是因为匿名类型并没有重载C#的相等操作符（==和!=）。因此当测试两个匿名对象是否相等时，使用“==”（而不是Equals()方法）将比较两者的引用，而不是指向的内容。

最后的条件测试（我们比较两个对象的类型名是否相同）的结果表明两个对象的类型是一样的（都为<>f_AnonymousType0`3），这是因为firstCar和secondCar拥有相同的属性（Color、Make和CurrentSpeed）。

这就演示了一个重要且微妙的一点，只有当匿名类型包含匿名类型的唯一名字时，编译器才会生成一个新的类定义。因此，如果我们在同一程序集中声明两个相同的匿名类型（同样，也就是相同的名字），编译器只会生成一个匿名类型的定义。

11.5.5 包含匿名类型的匿名类型

我们可以创建一个由匿名类型组成的匿名类型。例如，你需要构建一个购买订单，它包含时间戳、价格和被购买的汽车。下面是一个新的匿名类型（稍微有点复杂），用于表示这个实体：

```
// 创建一个由匿名类型组成的匿名类型
var purchaseItem = new {
  TimeBought = DateTime.Now,
  ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
  Price = 34.000};

ReflectOverAnonymousType(purchaseItem);
```

现在，你应该十分了解定义匿名类型的语法了，但可能还不清楚这种新语言特性的主要用处在哪里。其实，我们应该谨慎地使用匿名类型，尤其在使用LINQ技术时（第12章将会介绍）。你永远不要因为匿名类型的出现而放弃使用强类型的类或者结构。要知道，匿名类型本身有很多的限制。

- 你并没有控制匿名类型的名称。
- 匿名类型继承System.Object。
- 匿名类型的字段和属性总是只读的。
- 匿名类型不支持事件、自定义方法、自定义操作符和自定义重写。
- 匿名类型是隐式封闭的（sealed）。
- 匿名类型的实例创建只使用默认构造函数。

但是，在使用LINQ技术编程时，我们可能需要快速构建一个实体的形状而不需要定义其功能，这时你会发现匿名类型的好处。

源代码 AnonymousTypes项目的源代码位于Chapter 11子目录下。

11.6 指针类型

现在介绍本章最后一个话题，它可能是众多C#特性中使用频率最低的。

说明 在下面的示例中，假设你有C/C++指针操作基础（如果没有相关背景，可以完全跳过这个主题）。再次强调，使用指针并不是编写大多数C#应用程序的常见任务。

在第4章中，我们学到了.NET平台定义了两种主要数据类别：值类型和引用类型。实际上还有第三种：指针类型。要使用指针类型，系统为我们提供了特定操作符和关键字，可以绕开CLR的内存管理机制，自己处理（如表11-2所示）。

表11-2 指针相关的C#操作符和关键字

操作符/关键字	作 用
`*`	该操作符用于创建一个指针变量（也就是一个表示直接内存位置的变量）。和在C/C++中一样，同样的操作符用于指针间接寻址
`&`	该操作符用于获取内存中变量的地址
`->`	该操作符用于访问一个由指针表示的类型的字段（C#点操作符的不安全版本）
`[]`	在不安全的上下文中，[]操作符允许我们索引由指针变量指向的位置（回顾C/C++中指针变量和[]操作符的相互影响）
`++`，`--`	在不安全的上下文中，递增和递减操作符可用于指针类型
`+`，`-`	在不安全的上下文中，加减操作符可用于指针类型
`==`，`!=`，`<`，`>`，`<=`，`=>`	在不安全的上下文中，比较和相等操作符可用于指针类型
`stackalloc`	在不安全的上下文中，stackalloc关键字可用于直接在栈上分配C#数组
`fixed`	在不安全的上下文中，fixed关键字可用于临时固定一个变量以使它的地址可被找到

在深入细节之前，我将指出需要使用指针类型的情况很少。尽管C#允许深入指针操作层面，但要知道.NET运行库对我们的意图一无所知。如果我们对一个指针操作不当，就要自己为此负责。既然有上面这些警告，还有什么场合需要使用指针类型呢？有两种情形。

- 要绕过CLR管理直接操作指针以优化应用程序的特定部分。
- 要调用基于C的.dll或调用需要指针作为参数的COM服务器。即使在这种情况下，为了支持System.IntPtr类型和System.Runtime.InteropServices.Marshal类型，也可以不使用指针类型。

在确实需要使用这项C#语言特性的情况下，我们应通过允许项目支持不安全代码，告知C#编译器（csc.exe）我们的意图。在使用C#命令行编译器（csc.exe）时，仅需提供如下的/unsafe标志作为参数即可：

```
csc /unsafe *.cs
```

在Visual Studio中，需要访问Properties页并在Build选项卡中启用Allow Unsafe Code选项（如图11-2所示）。首先创建一个新的控制台应用程序UnsafeCode，它支持不安全代码，然后确保它启用了这项设置。

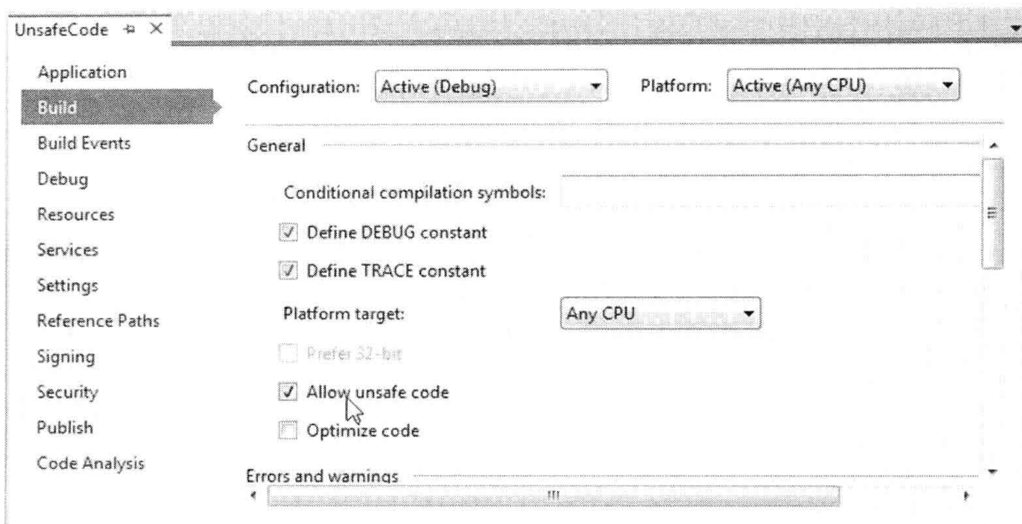


图11-2 使用Visual Studio启用不安全代码

11.6.1 unsafe关键字

当要使用C#中的指针时，必须使用unsafe关键字特别声明一个代码区块为不安全代码。不标记unsafe的任何代码都自动按“安全”代码对待。例如，下面的Program类在安全的Main()方法中声明了一组不安全的代码：

```
class Program
{
  static void Main(string[] args)
  {
    unsafe
    {
      // 在此处理指针类型
    }

    // 不能在此处理指针类型
  }
}
```

除了声明代码块为不安全代码外，还可构建“不安全的”结构、类、类型成员和参数。下面是截取的部分示例（不必在当前项目中声明Node和Node2）：

```
// 整个结构都是不安全的，仅可用于unsafe上下文中
unsafe struct Node
{
  public int Value;
  public Node* Left;
  public Node* Right;
}

// 这个结构是安全的，但Node2* 成员不安全
// 从技术上来讲，可以在unsafe上下文之外访问"Value"
```

```
// 但不能在unsafe上下文之外访问"Left"或"Right"
public struct Node2
{
  public int Value;

  // 这些只能在unsafe上下文中访问
  public unsafe Node2* Left;
  public unsafe Node2* Right;
}
```

静态方法或实例方法均可被标记为不安全。举个例子，假定你知道一个指定静态方法将使用指针逻辑。为确保该方法仅可自不安全上下文调用，可以按如下代码定义该方法：

```
unsafe static void SquareIntPointer(int* myIntPointer)
{
  // 将这个值平方后以供测试
  *myIntPointer *= *myIntPointer;
}
```

这个构造要求调用者这样调用SquareIntPointer()方法，如下所示：

```
static void Main(string[] args)
{
  unsafe
  {
    int myInt = 10;

    // 因为是在不安全上下文中
    SquareIntPointer(&myInt);
    Console.WriteLine("myInt: {0}", myInt);
  }

  int myInt2 = 5;

  // 编译器错误！必须在不安全上下文
  SquareIntPointer(&myInt2);
  Console.WriteLine("myInt: {0}", myInt2);
}
```

如果不想强制调用者将调用封装进不安全上下文，可以利用unsafe关键字修改Main()。此时，将编译如下代码：

```
unsafe static void Main(string[] args)
{
  int myInt2 = 5;
  SquareIntPointer(&myInt2);
  Console.WriteLine("myInt: {0}", myInt2);
}
```

运行该Main()方法，将得到如下所示的输出结果：

```
myInt: 25
```

11.6.2 *和&操作符

建立了不安全上下文之后，可以使用*操作符构建数据类型的指针，使用&操作符获取被指向的内存的地址。和C、C++不同，在C#中，*操作符仅被用于底层类型，而不是每个指针变量名的前缀。举

个例子，下列代码声明了两个变量，分别说明了声明整型变量指针的正确方法和错误方法：

```
// 错误！在C#中这是不合法的
int *pi, *pj;

// 正确！这才是C#的用法
int* pi, pj;
```

思考下列的不安全方法：

```
unsafe static void PrintValueAndAddress()
{
  int myInt;

  // 定义一个整数指针并将myInt的地址分配给它
  int* ptrToMyInt = &myInt;

  // 使用指针间接寻址为myInt赋值
  *ptrToMyInt = 123;

  // 输出一些状态
  Console.WriteLine("Value of myInt {0}", myInt);
  Console.WriteLine("Address of myInt {0:X}", (int)&ptrToMyInt);
}
```

11.6.3 不安全（与安全）交换功能

当然，给本地变量声明指针只是进行简单赋值（如在上面示例中所示），这没有必要而且没有用处。为便于说明更为实际的不安全代码示例，假定要使用指针算法实现一个交换功能：

```
unsafe public static void UnsafeSwap(int* i, int* j)
{
  int temp = *i;
  *i = *j;
  *j = temp;
}
```

是不是很像C语言？不过，大家通过前面的学习，应该意识到我们可以使用C# ref关键字写出下列交换算法的安全版本：

```
public static void SafeSwap(ref int i, ref int j)
{
  int temp = i;
  i = j;
  j = temp;
}
```

两个方法的功能都是一样的，再次强调，直接指针操作并不是C#下的必要任务。下面是使用安全Main()和不安全上下文的调用逻辑：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Calling method with unsafe code *****");

  // 以备交换的值
  int i = 10, j = 20;
```

11

```
    // 安全交换两个值
    Console.WriteLine("\n***** Safe swap *****");
    Console.WriteLine("Values before safe swap: i = {0}, j = {1}", i, j);
    SafeSwap(ref i, ref j);
    Console.WriteLine("Values after safe swap: i = {0}, j = {1}", i, j);

    // 不安全交换两个值
    Console.WriteLine("\n***** Unsafe swap *****");
    Console.WriteLine("Values before unsafe swap: i = {0}, j = {1}", i, j);

    unsafe { UnsafeSwap(&i, &j); }

    Console.WriteLine("Values after unsafe swap: i = {0}, j = {1}", i, j);
    Console.ReadLine();
  }
```

11.6.4 通过指针访问字段

假定定义了一个Point结构的指针：

```
struct Point
{
  public int x;
  public int y;

  public override string ToString()
  {
    return string.Format("({0}, {1})", x, y);
  }
}
```

如果声明一个Point类型的指针，就需要使用指针字段访问操作符(->)来访问公共成员。如表11-2所示，这是标准（安全）的点操作符的不安全版本。事实上，使用指针间接寻址操作符（*）来解除指针的引用，使其也可以使用点操作符访问字段。看看下面的不安全方法：

```
unsafe static void UsePointerToPoint()
{
  // 通过指针访问成员
  Point point;
  Point* p = &point;
  p->x = 100;
  p->y = 200;
  Console.WriteLine(p->ToString());

  // 通过指针间接寻址访问成员
  Point point2;
  Point* p2 = &point2;
  (*p2).x = 100;
  (*p2).y = 200;
  Console.WriteLine((*p2).ToString());
}
```

11.6.5 stackalloc关键字

在不安全上下文中，可能需要声明一个直接从调用栈分配内存（不受制于.NET垃圾收集器）的本

地变量。C#提供了与C运行库函数_alloca等效的stackalloc关键字来满足这个要求。简单示例如下：

```
unsafe static void UnsafeStackAlloc()
{
  char* p = stackalloc char[256];
  for (int k = 0; k < 256; k++)
    p[k] = (char)k;
}
```

11.6.6 使用fixed关键字固定类型

在前一个示例中看到，通过使用stackalloc关键字，在不安全上下文中分配一大块内存非常方便。从这个操作的本意来说，由于被分配的内存是从栈中获得的，当分配方法返回的时候，被分配的内存立即被清理。再思考一个更为复杂的示例。在我们讨论->操作符的时候，创建了名为Point的值类型。和其他值类型一样，分配的内存在执行代码块终止后将出栈。为了讨论方便，假定Point被定义为一个引用类型：

```
class PointRef  // 重命名和重新定义类型后的<=
{
  public int x;
  public int y;
  public override string ToString()
  {
    return string.Format("({0}, {1})", x, y);
  }
}
```

要注意，如果调用者声明了一个Point类型的变量，内存将被分配在垃圾收集堆上。这时严重的问题出现了，如果一个不安全上下文要与这个对象（或这个堆上的任何对象）交互，又该怎么办呢？由于垃圾收集可随时发生，设想一下恰好在清理堆的时候访问Point成员，该是多么痛苦啊！理论上来讲，不安全上下文能够进行交互的成员包括：不再允许访问的成员或堆上经清扫后已被重置的成员，但这很明显有问题。

11

为了将不安全上下文内存中的引用类型变量固定，C#提供了fixed关键字。fixed语句设置指向托管类型的指针并在代码执行过程中固定该变量。由于垃圾收集器将在不可预知的情况下重置变量，没有fixed的话，托管变量的指针将没有多大用处。（事实上，除了在fixed语句中，C#编译器不允许设置指向托管变量的指针。）

不过，如果创建了一个Point类型并想与它的成员交互，必须写下列代码，否则将收到编译器错误：

```
unsafe public static void UseAndPinPoint()
{
  PointRef pt = new PointRef ();
  pt.x = 5;
  pt.y = 6;

  // 在适当位置固定pt以免GC除去
  fixed (int* p = &pt.x)
  {
    // 在此使用int*变量
  }

  // pt现未被固定，在该方法完成后可被GC清除
```

```
  Console.WriteLine ("Point is: {0}", pt);
}
```

简而言之，fixed关键字允许我们构建锁定内存中引用变量的语句，这样一来，在该语句的执行过程中，该变量地址保持不变。是的，任何时候在不安全代码上下文中与引用类型交互，都要固定该引用。

11.6.7　sizeof关键字

最后一个不安全相关的C#关键字是sizeof。和C/C++相比而言，C#的sizeof关键字用来获取值类型（不是引用类型）的字节大小，它仅可被用在不安全上下文中。与基于C的非托管API交互时，这种能力非常有用。它的用法很简单：

```
unsafe static void UseSizeOfOperator()
{
  Console.WriteLine("The size of short is {0}.", sizeof(short));
  Console.WriteLine("The size of int is {0}.", sizeof(int));
  Console.WriteLine("The size of long is {0}.", sizeof(long));
}
```

sizeof可计算任何System.ValueType派生实体的字节数，也可以获取自定义结构的大小。例如，假设将Point结构传入sizeof：

```
unsafe static void UseSizeOfOperator()
{
...
  Console.WriteLine("The size of Point is {0}.", sizeof(Point));
}
```

源代码　UnsafeCode项目的源代码位于Chapter 12子目录下。

本章介绍了C#编程语言中一些更加高级的特性。为了证明我们是在一条船上，我必须再次说明，大多数.NET项目都不需要直接使用这些特性（特别是指针）。然而我们将在下一章看到，在使用LINQ API时，一些特性是非常有用的，特别是扩展方法和匿名类型。

11.7　小结

本章的目的在于加深大家对C#编程语言的理解。首先，我们讨论了各种高级类型的构造技巧（索引器方法、重载操作符和自定义转换例程）。

接下来，我们学习了扩展方法和匿名类型的作用。下一章中你将看到，这些特性在LINQ相关的API中十分有用（当然你可以在任何地方使用这些特性，并且它们也十分有用）。匿名方法可以快速构建类型的“轮廓”，扩展方法可以为类型添加新的功能（不需要创建子类）。

最后学习了如何使用原始的指针类型，同时介绍了一些不为人知的关键字（sizeof和unsafe等）。绝大多数C#应用程序都没必要使用指针类型。

第 12 章 LINQ to Object

不管使用.NET平台创建何种应用程序，都需要在执行时访问某种形式的数据，如XML文件、关系数据库、内存中的集合、基元数组等。数据是无处不在的。过去，根据数据位置的不同，程序员需要使用不同且不相关的API。.NET 3.5引入了语言集成查询（LINQ），它提供了一种简明的、对称的、强类型的方式访问各种各样的数据存储。本章将开始研究LINQ，首先关注的是LINQ to Object。

在介绍LINQ to Object之前，本章将快速浏览C#中与LINQ相关的关键语言结构。在学习本章时，你会发现隐式类型本地变量、对象初始化语法、Lambda表达式、扩展方法以及匿名类型都是非常有用的（当然我指的不是为了用而用）。

回顾了支持的基础设施之后，本章余下的内容将向你展示LINQ编程模型和它在.NET平台中的作用。还有，你将开始学习查询操作符和查询表达式的作用，这些东西允许你定义语句来对数据源进行查询，获取所请求的结果集。与此同时，你将构建许多LINQ例程，与包含在数组和集合类型（包括泛型和非泛型）里的数据交互。此外，你还会学到程序集、命名空间和表示LINQ to Object API的类型。

说明 本章所涵盖的内容是其他LINQ技术的基础，如LINQ to XML（参见第24章）、并行LINQ（参见第19章）和LINQ to Entity（参见第23章）。

12.1 LINQ 特有的编程结构

从宏观上看，LINQ可以理解为直接嵌入C#语法的强类型查询语言。使用LINQ，可以构建与数据库SQL查询类似的表达式。但LINQ查询可以用于多种数据存储，甚至与关系型数据库完全无关的存储。

说明 尽管LINQ查询看起来与SQL查询很像，但语法却并不相同。实际上，很多LINQ查询的格式看上去与类似的数据库查询完全相反！如果要将LINQ直接映射为SQL，你肯定会很沮丧。为了保持清醒，我建议你尽量把LINQ查询看成是独特的语句，它只是“碰巧看上去”像SQL而已。

当.NET 3.5平台首次引入LINQ时，C#和VB语言为了支持LINQ技术集，扩展了大量新的编程结构。其中C#使用了如下与LINQ相关的特性：

- ❑ 隐式类型本地变量
- ❑ 对象/集合初始化语法
- ❑ Lambda表达式
- ❑ 扩展方法
- ❑ 匿名类型

本书已经在不同章节中详细介绍了这些特性。但为了确保已经掌握了这些特性，我们先对它们进行一次快速的浏览。

12.1.1 隐式类型本地变量

我们在第3章中学习了C#的var关键字。它允许你定义不显式指定实际数据类型的本地变量。不过由于编译器将根据初始值推断其数据类型，所以该变量仍然是强类型的。回忆一下第3章中的这个示例：

```
static void DeclareImplicitVars()
{
  // 隐式类型本地变量
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";

  // 打印实际类型
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
  Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
  Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

在使用LINQ时，这项语言特性十分有帮助，并且常常是强制性的。在本章中你将看到，许多LINQ查询返回的序列只有在编译时才能确定其数据类型。由于只有在应用程序编译之后才能知道其实际的数据类型，所以显然无法显式地声明变量。

12.1.2 对象和集合初始化语法

第5章介绍了对象初始化语法的作用，它允许我们在创建类或结构变量的同时设置其属性。最终得到的是紧凑的（且易读的）语法和创建好的对象。同样回忆第9章，C#语言还允许我们使用非常类似的语法初始化集合对象。考虑下面的代码段，它使用集合初始化语法填充Rectangle对象的List<T>，其中每个Rectangle包含两个Point对象，表示(x, y)坐标：

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
  new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                 BottomRight = new Point { X = 200, Y = 200}},
  new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                 BottomRight = new Point { X = 100, Y = 100}},
  new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                 BottomRight = new Point { X = 90, Y = 75}}
};
```

尽管没有要求必须使用集合/对象初始化语法，但这样可以得到更紧凑的代码结构。此外，当它们

与隐式类型本地变量相结合时，可以声明匿名类型，这在创建LINQ投影时也是十分有用的。本章稍后将学习LINQ投影。

12.1.3 Lambda表达式

我们在第10章中充分介绍了C# Lambda操作符（=>），它可以用来构建Lambda表达式，并且在调用以强类型的委托作为参数的方法时，也十分有用。Lambda大大简化了.NET委托的使用，减少了需要手工输入的代码。Lambda表达式的用法如下：

```
(ArgumentsToProcess) => {StatementsToProcessThem}
```

在第10章中，我使用了3种不同的方法与泛型List<T>类的FindAll()方法进行交互。在使用原始的Predicate<T>委托和C#匿名方法之后，我们使用Lambda表达式得到了如下所示的（极其简洁的）迭代：

```
static void LambdaExpressionSyntax()
{
  // 建一个整数列表
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // C# Lambda表达式
  List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);

  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
{
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

在使用LINQ对象模型时，Lambda是非常有用的。稍后你会发现，C# LINQ查询操作符是调用System.Linq.Enumerable类中方法的简便方式。这些方法通常都使用委托（如Func<>委托）作为参数，用来处理数据以生成正确的结果集。使用Lambda表达式可以精简代码，并且让编译器来推断实际的委托。

12.1.4 扩展方法

C#扩展方法不使用子类就能够向已知类中添加新的功能。同样，它还可以向不能有子类的密封类和结构中添加新的功能。回忆第11章，在编写扩展方法时，第一个参数必须使用this限定符，用来表示被扩展的类型。扩展方法只能定义在静态类中，并且必须使用static关键字声明为静态方法。例如：

```
namespace MyExtensions
{
  static class ObjectExtensions
  {
  // 为System.Object定义扩展方法
    public static void DisplayDefiningAssembly(this object obj)
    {
      Console.WriteLine("{0} lives here:\n\t->{1}\n", obj.GetType().Name,
        Assembly.GetAssembly(obj.GetType()));
    }
  }
}
```

要使用该扩展，应用程序必须导入定义扩展的命名空间（可能需要设置到外部程序集的引用）。然后，引入定义的命名空间，编写如下代码：

```
static void Main(string[] args)
{
// 由于任何类型都扩展自System.Object，因此所有类和结构都可以使用该扩展方法
  int myInt = 12345678;
  myInt.DisplayDefiningAssembly();

  System.Data.DataSet d = new System.Data.DataSet();
  d.DisplayDefiningAssembly();
  Console.ReadLine();
}
```

在使用LINQ时，很少需要手工构建扩展方法。但在创建LINQ查询表达式时，你实际上是在使用微软已经定义的大量扩展方法。其实，每个C# LINQ查询操作符都是一种简便记法，用来调用定义在System.Linq.Enumerable工具类中的扩展方法。

12.1.5　匿名类型

最后一个要快速浏览的C#语言特性是匿名类型，这已经在第11章中进行了完整的介绍。该特性可以快速建立数据的“结构”，编译器将根据名称/值对的集合在编译时生成新的类。该类型是基于值的语义构建的，因此System.Object中的每个虚方法都要重写。要定义一个匿名类型，可以声明一个隐式类型变量，并使用对象初始化语法指定数据的结构：

```
// 创建一个由匿名类型组成的匿名类型
var purchaseItem = new {
  TimeBought = DateTime.Now,
  ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
  Price = 34.000};
```

LINQ经常使用匿名类型在运行时投影新的数据形式。例如，假设我们有一个Person对象的集合，希望使用LINQ获得年龄和社会保险号码的信息。使用LINQ投影，可以使编译器生成新的包含这些信息的匿名类型。

12.2　LINQ 的作用

我们介绍完了能够使LINQ如此神奇的C#语言特性。但是，LINQ为什么如此重要呢？作为软件开发人员，很难否认我们很大一部分编程时间都是花在获取和操作数据上的。说到“数据”，大概就会自然而然地想到包含在关系数据库里的信息。另一个很流行的数据格式是XML文档，例如*.config文件、保存在本地的DataSet、从WCF服务返回的内存中的数据等。

除了这两个常见的信息来源外，还能在许多地方找到数据。例如，如果你有个List<T>泛型，内含300个整数，你要获取满足某个给定条件（例如，仅奇数或偶数，仅质数，仅大于50的不重复数字等）的子集。或者，你也许要利用反射API，从一个Type数组中只想得到从一个特定父类继承而来的每个类型的元数据描述。显而易见，数据随处可见。

在.NET 3.5之前的版本中，与特定类型的数据打交道时，程序员要使用不同的API。例如，考虑一下表12-1，它列出了操作各种数据类型的常见API（我确定你可以想到很多其他示例）。

表12-1 操作各种数据的方式

我们想要的数据	得到这些数据的方式
关系数据	System.Data.dll和System.Data.SqlClient.dll等
XML文档数据	System.Xml.dll
元数据表	System.Reflection命名空间
对象集合	System.Array和System.Collections/System.Collections.Generic命名空间

当然，这些操作数据的方法一点也没问题。实际上，你肯定会直接使用ADO.NET、XML命名空间、反射服务以及各种集合类型。但是，基本的问题在于，这些API的每一种本身只是孤岛，提供极少的集成方式。没错，例如，你能把ADO.NET DataSet保存成XML，然后通过System.Xml命名空间来操作。但尽管如此，目前的数据操作依然相当不对称。

LINQ（语言级集成查询）API的意图是提供一种统一且对称的方式，让程序员在广义的数据上得到和操作"数据"。通过使用LINQ，我们能够在C#编程语言内直接创建被称为查询表达式（query expression）的实体。这些查询表达式是基于许多查询操作符（query operator）的，而且是有意设计成类似SQL表达式的。

但不同之处在于，查询表达式可以被用来与许多种数据类型交互，即使是那些与关系数据库毫无关系的数据。具体来说，LINQ是用来描述数据访问总体方式的术语。根据LINQ查询应用的场景，可以分为以下几个部分。

- LINQ to Object：针对数组和集合使用的LINQ查询。
- LINQ to XML：使用LINQ来操纵和查询XML文档。
- LINQ to DataSet：针对ADO.NET DataSet对象使用的LINQ查询。
- LINQ to Entity：对ADO.NET Entity Framework（EF）API使用的LINQ查询。
- Parallel LINQ（PLINQ）：并行处理LINQ查询返回的数据。

可以肯定的是，微软致力于在.NET编程环境中集成对于LINQ的支持。如今，LINQ已经成为.NET基础类库、托管语言和Visual Studio不可分割的一部分。

12.2.1 LINQ表达式是强类型的

需要特别指出的是，LINQ查询表达式（跟传统的SQL语句有所不同）是强类型的。所以，C#编译器会让我们老老实实地保证这些表达式在语法上是合法的。另外需要注意的是，查询表达式在利用它们的程序集中拥有对应的元数据表示，因为C# LINQ查询表达式总是构建丰富的基本对象模型。像Visual Studio这样的工具可以使用这个元数据来提供诸如智能感知、自动完成等有用的功能。

12.2.2 核心LINQ程序集

第2章提到过，Visual Studio的New Project对话框中有一个选项，我们可以使用下拉框来选择希望编译针对的.NET平台版本。如果选择针对.NET 3.5或者更高版本编译，每一个项目模板都会自动引用LINQ程序集（它们可以通过Solution Explorer查看）。表12-2描述了核心LINQ程序集的作用。但是在本书其他部分，你会遇到其他LINQ库。

表12-2 核心LINQ程序集

程 序 集	内 容 描 述
System.Core.dll	定义了代表核心LINQ API的类型。如果你想使用任何LINQ API（包括LINQ to Object），这是你必须访问的程序集
System.Data.DataSetExtensions.dll	定义了许多类型来将ADO.NET类型融入LINQ编程范式（LINQ to DataSet）
System.Xml.Linq.dll	提供了使用LINQ处理XML文档数据所需的功能（LINQ to XML）

为了使用LINQ to Object，我们必须确保每个包含LINQ查询的代码文件都导入System.Linq命名空间（主要定义在System.Core.dll中）。否则将会出现很多问题。按经验来说，将会出现类似下面的编译器错误：

```
Error 1 Could not find an implementation of the query pattern for source type 'int[]'.
'Where' not found.  Are you missing a reference to 'System.Core.dll' or a using directive
for 'System.Linq'?
```

而此时你的C#文件中很可能没有下面的using指令（凭经验来说）：

```
using System.Linq;
```

12.3 将 LINQ 查询应用于原始数组

为开始研究LINQ to Object，让我们先建立一个应用程序，其中会将LINQ查询应用于各种数组对象。创建一个名为LinqOverArray的控制台应用程序，在Program类型里定义一个名为QueryOverStrings()的静态辅助方法。在这个方法里，创建一个字符串数组，内含大约6个你喜欢的字符串（这里，我列出了几个我目前试图完成的电视游戏）。要确保至少有两项包含数字，一些项包含空格。

```
static void QueryOverStrings()
{
  // 假定我们有一个字符串数组
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                "Fallout 3", "Daxter", "System Shock 2"};
}
```

最后，更新Main()来调用QueryOverStrings()：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with LINQ to Objects *****\n");
  QueryOverStrings();
  Console.ReadLine();
}
```

当你有任意一组数据时，你经常会根据给定的要求提取一个数据项子集。也许你只想得到内含数字的子项（例如，System Shock 2、Uncharted 2和Fallout 3），或者长度大于某个数目的子项，或者不含空格的子项（例如，Morrowind或Daxter）。虽然你可以使用System.Array类型的成员方法来完成这样的工作，但肯定会费点劲，而LINQ查询表达式却能极大地简化这个过程。

再假设你只想从数组中得到包含空格的项，并让它们按照字母顺序排序，我们可以建立下面这样的查询表达式：

```
static void QueryOverStrings()
{
  // 假定我们有一个字符串数组
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // 构建一个查询表达式，来代表数组中有一个空格的项
  IEnumerable<string> subset = from g in currentVideoGames
                      where g.Contains(" ") orderby g select g;

  // 输出结果
  foreach (string s in subset)
    Console.WriteLine("Item: {0}", s);
}
```

注意，刚建立的查询表达式利用了from、in、where、orderby和select这几个LINQ查询操作符。我们不久就会深入探究查询表达式的完整句法，但即使现在，你应该也能将这个语句分析成“把那些名字中包含空格的currentVideoGames中的项按字母表顺序排序”。

在这里，符合这个条件的项命名为“g”（“游戏”的英文game中的首字母）。但是，任何合法的C#变量名都可以用在这里，例如：

```
IEnumerable<string> subset = from game in currentVideoGames
                             where game.Contains(" ") orderby
                             game select game;
```

最后请注意，“结果集”是由一个实现了IEnumerable<T>泛型版本的对象来表示的，这里T是System.String类型（我们毕竟是在查询一个字符串数组）。在得到结果集后，我们可以使用标准的foreach语句简单地输出每个字符串项。如果运行应用程序，得到的输出结果如下所示：

```
***** Fun with LINQ to Objects *****
Item: Fallout 3
Item: System Shock 2
Item: Uncharted 2
```

12

12.3.1 再一次，不使用LINQ

要知道，LINQ永远都不是强制使用的。如果不使用LINQ，而使用if语句、for循环之类的编程基元，也可以得到相同的结果。下面的方法能够产生和QueryOverStrings()方法相同的结果，只不过更加烦琐：

```
static void QueryOverStringsLongHand()
{
// 假定我们有一个字符串数组
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  string[] gamesWithSpaces = new string[5];
```

```
    for (int i = 0; i < currentVideoGames.Length; i++)
    {
      if (currentVideoGames[i].Contains(" "))
        gamesWithSpaces[i] = currentVideoGames[i];
    }

  // 排序
    Array.Sort(gamesWithSpaces);

  // 打印结果
    foreach (string s in gamesWithSpaces)
    {
      if( s != null)
        Console.WriteLine("Item: {0}", s);
    }
    Console.WriteLine();
  }
```

我知道上面的方法还可以进一步调整和简化，但无法改变一个事实，即用LINQ查询从数据源中提取数据子集要容易得多。只要创建了恰当的LINQ查询，C#编译器就可以为我们执行那些脏活累活，如构建嵌套循环、复杂的if/else逻辑、临时数据类型等。

12.3.2　反射LINQ结果集

现在假定Program类定义了另外一个辅助函数叫ReflectOverQueryResults()，该函数将输出LINQ结果集的种种细节（注意，方法参数是一个System.Object，用来处理各种结果集）：

```
static void ReflectOverQueryResults(object resultSet)
{
  Console.WriteLine("***** Info about your query *****");
  Console.WriteLine("resultSet is of type: {0}", resultSet.GetType().Name);
  Console.WriteLine("resultSet location: {0}",
    resultSet.GetType().Assembly.GetName().Name);
}
```

假定你在输出得到的子集之后，在QueryOverStrings()内调用了这个方法，如果你运行该应用的话，将看到这个子集实际上是一个OrderedEnumerable<TElement,Tkey>泛型类型的实例（是由OrderedSequence 2这样的CIL代码来表示的），该类型位于System.Core.dll程序集中：

```
***** Info about your query *****

resultSet is of type: OrderedEnumerable`2
resultSet location: System.Core
```

说明　很多表示LINQ结果的类型都被Visual Studio对象浏览器隐藏了。这些底层类型设计出来并不是要在你的应用程序中直接使用的。

12.3.3 LINQ和隐式类型本地变量

尽管当前示例程序可以相对容易地确定结果集是string对象的枚举（如IEnumerable<string>），但我猜你可能并不清楚subset实际上也是OrderedEnumerable<TElement, TKey>类型。

由于LINQ结果集可以用很多来自LINQ命名空间的类型来表示，定义确切的类型来接受结果集会是非常枯燥乏味的，因为在很多情形下底层的类型可能并不那么明显或者不能从代码库直接访问（有时这个类型是在编译时生成的）。

为进一步强调这一点，考虑一下这个在Program类中定义的另外一个辅助方法（假定你会在Main()方法中调用它）：

```
static void QueryOverInts()
{
  int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};

  // 只输出小于10的项
  IEnumerable<int> subset = from i in numbers where i < 10 select i;

  foreach (int i in subset)
    Console.WriteLine("Item: {0}", i);
  ReflectOverQueryResults(subset);
}
```

在这种情况下，subset变量是完全不同的实际类型。这时实现IEnumerable<int>接口的类型是低级别的WhereArrayIterator<T>类：

```
Item: 1
Item: 2
Item: 3
Item: 8

***** Info about your query *****
resultSet is of type: WhereArrayIterator`1
resultSet location: System.Core
```

由于LINQ查询提取的实际类型不是很明显，所以第一个示例中用IEnumerable<T>变量来表示查询结果，其中T为返回序列中的数据类型（string、int等）。但这仍然显得很笨重。更糟的是，由于IEnumerable<T>扩展了非泛型的IEnumerable接口，你还可以像下面这样获取LINQ查询的结果：

```
System.Collections.IEnumerable subset =
  from i in numbers where i < 10 select i;
```

幸好在处理LINQ查询时，隐式类型可以极大地简化代码：

```
static void QueryOverInts()
{
  int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};

  // 使用隐式类型
  var subset = from i in numbers where i < 10 select i;

  // 这里也使用隐式类型
```

```
    foreach (var i in subset)
      Console.WriteLine("Item: {0} ", i);
    ReflectOverQueryResults(subset);
  }
```

一般来说，在获取LINQ查询的结果集时，应该总是使用隐式类型。但要记住的是，在绝大多数情况下，真正的返回值是实现了泛型IEnumerable<T>接口的类型。

至于该类型究竟是什么（OrderedEnumerable<TElement, TKey>、WhereArrayIterator<T>等）是没有关系的，而且也没有必要知道。如上例所示，你可以在foreach结构中简单地使用var关键字对获取的数据进行迭代。

12.3.4 LINQ和扩展方法

虽然当前的例子不需要你编写任何扩展方法，但事实上，你却在背后无缝地使用了它们。回想一下，LINQ查询表达式可以用来在实现了泛型IEnumerable<T>接口的数据容器上做迭代操作。但是，.NET System.Array这个类（可以用来表示字符串数组和整数数组）并没有实现这个行为：

```
// System.Array类型看上去并没有实现查询表达式所需的正确接口
public abstract class Array : ICloneable, IList, ICollection,
  IEnumerable, IStructuralComparable, IStructuralEquatable

{
  ...
}
```

虽然System.Array并没有直接实现IEnumerable<T>接口，但它通过静态的System.Linq. Enumerable类类型间接地得到了该类型所需的功能，同时还得到了许多其他的与LINQ相关的成员。

这个工具类定义了许多泛型的扩展方法（像Aggregate<T>()、First<T>()、Max<T>()等），这些都是System.Array（和其他类型）在幕后获得的扩展方法。因此，如果你在currentVideoGames本地变量上应用点操作符的话，你会发现很多在System.Array类的正式定义里找不到的成员（参见图12-1）。

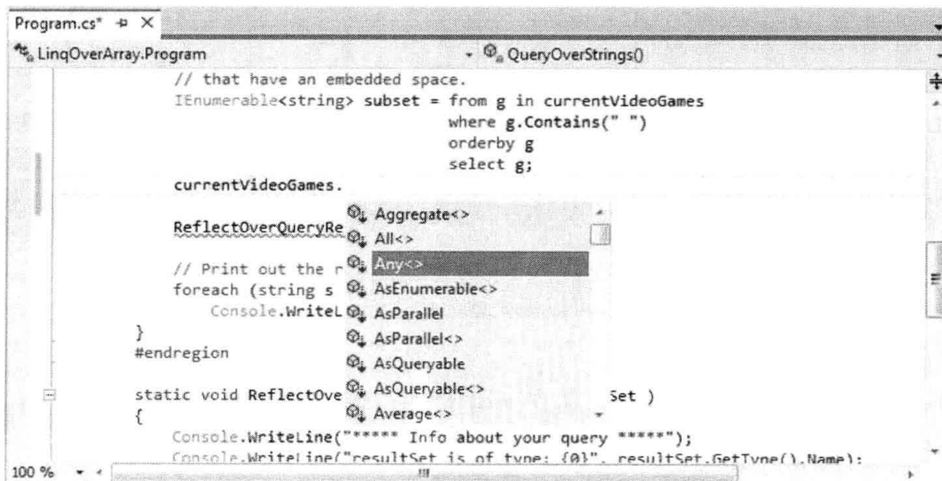


图12-1 System.Array类型已被System.Linq.Enumerable的成员扩充

12.3.5 延迟执行的作用

有关LINQ查询表达式另一个重要的地方是在我们迭代内容之前，它们不会真正进行运算。严格地说，这叫做延迟执行。这种方式的好处在于可以为相同的容器多次应用相同的LINQ查询，而始终可以获得最新的最好的结果。考虑对QueryOverInts()方法的如下更新：

```
static void QueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // 获取小于10的数
  var subset = from i in numbers where i < 10 select i;

  // LINQ语句在这里运算
  foreach (var i in subset)
    Console.WriteLine("{0} < 10", i);
  Console.WriteLine();

  // 修改数组中的一些数据
  numbers[0] = 4;

  // 再一次运算
  foreach (var j in subset)
    Console.WriteLine("{0} < 10", j);

  Console.WriteLine();
  ReflectOverQueryResults(subset);
}
```

再次运行该程序，将得到以下输出结果。注意，第二次对序列进行迭代时，会多出一个成员，因为你在数组头部添加了一个小于10的项：

```
1 < 10
2 < 10
3 < 10
8 < 10

4 < 10
1 < 10
2 < 10
3 < 10
8 < 10
```

Visual Studio 一个很有用的方面是，如果在运算LINQ查询之前设置断点，就可以在调试会话中查看内容。只需要把我们的鼠标光标放在LINQ结果集变量的位置（图12-2中的subset），就可以通过展开Result View选项来运算查询。

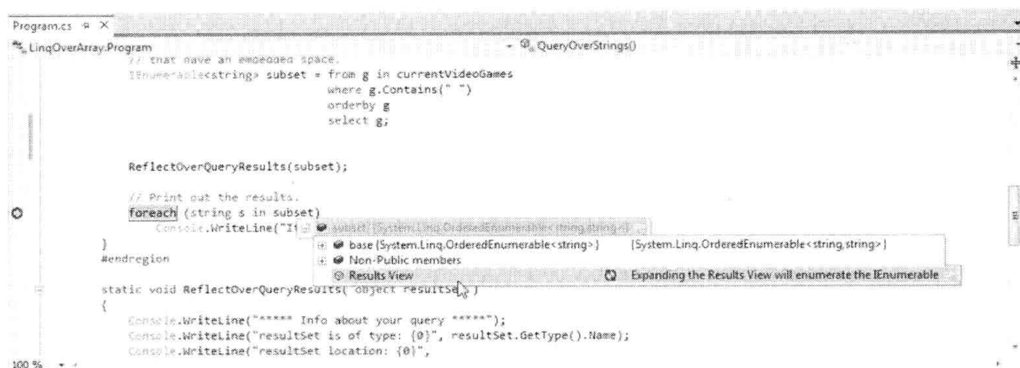


图12-2　调试LINQ表达式

12.3.6　立即执行的作用

如果希望在foreach逻辑外部运算LINQ表达式，可以调用由Enumerable类型定义的许多扩展方法来完成。Enumerable定义了诸如ToArray<T>()、ToDictionary<TSource, TKey>()以及ToList<T>()在内的许多扩展方法。在调用这些方法的同时将执行LINQ查询，以获取数据快照。然后，这些数据快照就可以独立进行操作了：

```
static void ImmediateExecution()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  // 立即获取数据为int[]
  int[] subsetAsIntArray =
    (from i in numbers where i < 10 select i).ToArray<int>();

  // 立即获取数据为List<int>
  List<int> subsetAsListOfInts =
    (from i in numbers where i < 10 select i).ToList<int>();
}
```

注意，整个LINQ表达式用圆括号括起来，这样就能将它强制转换为正确的实际类型来调用Enumerable的扩展方法。

第9章提到过，C#编译器可以准确检测泛型项的类型参数，我们不需要指定类型参数。因此，我们可以按如下代码调用ToArray<T>()（或ToList<T>()）：

```
int[] subsetAsIntArray =
  (from i in numbers where i < 10 select i).ToArray();
```

当你要对外部调用者返回LINQ查询时，立即执行的好处就显而易见了。这正是本章的下一个话题。

源代码　LinqOverArray项目的源代码位于Chapter 12子目录下。

12.4 返回 LINQ 查询的结果

你可以在类（或结构）中定义一个字段，使其值为LINQ查询的结果。但这样就不能使用隐式类型了（因为var关键字不能用于字段），并且LINQ查询的目标也不能是实例数据，因此必须是静态的。由于这些限制，我们通常不会编写以下代码：

```
class LINQBasedFieldsAreClunky
{
  private static string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // 不能使用隐式类型！必须知道subset的类型
  private IEnumerable<string> subset = from g in currentVideoGames
    where g.Contains(" ") orderby g select g;

  public void PrintGames()
  {
    foreach (var item in subset)
    {
      Console.WriteLine(item);
    }
  }
}
```

LINQ查询通常都定义在方法或属性作用域内。此外，为了简化编程，用于保存结果集的变量往往用var关键字声明为隐式类型本地变量。现在，回忆第3章所描述的内容，隐式类型变量不能用来定义类和结构的参数、返回值或字段。

那么，如何向外部调用者返回查询结果呢？答案是视情况而定。如果结果集由强类型数据组成（如字符串数组或Car的List<T>），可以使用恰当的IEnumerable<T>或IEnumerable类型（同样，IEnumerable<T>扩展自IEnumerable），而不用var关键字。下面的示例来自新建的名为LinqRetValues的Console Application：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** LINQ Transformations *****\n");
    IEnumerable<string> subset = GetStringSubset();

    foreach (string item in subset)
    {
      Console.WriteLine(item);
    }

    Console.ReadLine();
  }

  static IEnumerable<string> GetStringSubset()
  {
    string[] colors = {"Light Red", "Green",
      "Yellow", "Dark Red", "Red", "Purple"};

    // 注意，subset是IEnumerable<string>兼容的对象
```

12

```
        IEnumerable<string> theRedColors = from c in colors
           where c.Contains("Red") select c;

        return theRedColors;
      }
    }
```

输出结果如下：

```
Light Red
Dark Red
Red
```

通过立即执行返回LINQ结果

由于GetStringSubset()的返回值和它内部的LINQ查询都是强类型的，所以该示例可以如期运行。如果使用var关键字定义subset变量，则只有在该方法仍然定义返回IEnumerable<string>（并且隐式类型本地变量与指定的返回类型兼容）时，才允许返回值。

你可以通过立即执行来避免操作IEnumerable<T>带来的不便。例如，如果你将序列转换为强类型数组，就可以简单地返回string[]，而不必返回IEnumerable<string>。以下这个Program类的新方法实现了这样的功能：

```
static string[] GetStringSubsetAsArray()
{
  string[] colors = {"Light Red", "Green",
    "Yellow", "Dark Red", "Red", "Purple"};

  var theRedColors = from c in colors
    where c.Contains("Red") select c;

  // 将结果映射为数组
  return theRedColors.ToArray();
}
```

这样，调用者很乐意不知道得到的结果是否来自LINQ查询，它们只需要简单地操作string数组即可。如下面的代码：

```
foreach (string item in GetStringSubsetAsArray())
{
   Console.WriteLine(item);
}
```

在返回LINQ投影的结果时，立即执行也是十分关键的。本章稍后将讨论该话题。不过接下来我们先来看看如何对泛型和非泛型集合对象使用LINQ查询。

源代码 LinqRetValues项目的源代码位于Chapter 12子目录下。

12.5 将 LINQ 查询应用到集合对象

除了从简单的数据数组里抽出结果以外，LINQ查询表达式也可以操作System.Collections.Generic命名空间中的成员类型的数据，例如List<T>类型。创建一个名为LinqOverCustomObjects的新控制台项目，定义一个基本的Car类，内含当前的速度、颜色、型号和昵称，代码如下：

```
class Car
{
  public string PetName {get; set;}
  public string Color {get; set;}
  public int Speed {get; set;}
  public string Make {get; set;}
}
```

现在，在你的Main()方法里定义一个类型为Car的局部List<T>变量，使用新的对象初始化语法给这个表填充几个新的Car对象：

```
static void Main(string[] args)
{
  Console.WriteLine("***** LINQ over Generic Collections *****\n");

  // 生成一系列Car对象
  List<Car> myCars = new List<Car>() {
    new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
    new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
    new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
    new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
    new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
  };

  Console.ReadLine();
}
```

12.5.1 访问包含的子对象

对泛型容器使用LINQ查询与简单的数组没有什么区别，因为LINQ to Object可以用于任何实现了IEnumerable<T>的类型。我们这一次的目标是构建一个查询表达式，选择myCars列表中速度大于55的Car对象。

在得到子集后，通过调用PetName属性打印每个Car对象的名称。假设Main()中调用了如下的辅助方法（以List<Car>作为参数）：

```
static void GetFastCars(List<Car> myCars)
{
// 找到List<>中所有Speed大于55的Car对象
  var fastCars = from c in myCars where c.Speed > 55 select c;

  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

12

注意，我们的查询表达式只从List<T>里抓取Speed属性大于55的那些项。如果运行程序，我们会发现只有Henry和Daisy两项符合搜索条件。

如果我们要建一个复杂的查询，也许会想只找那些速度超过90的宝马车。要这么做的话，只要使用C#的&&操作符建一个复杂的布尔语句即可：

```
static void GetFastBMWs(List<Car> myCars)
{
  // 找到最快的宝马车
  var fastCars = from c in myCars where c.Speed > 90 && c.Make == "BMW" select c;
  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

在这个情形下，输出的昵称只有Henry。

12.5.2　将LINQ查询应用于非泛型集合

回想一下，LINQ的查询操作符是设计用于任何实现了IEnumerable<T>接口的类型的，无论是直接地还是通过扩展方法间接实现的。鉴于System.Array已经具备了这些必需的基础结构，System.Collections中传统的非泛型容器类却没有这些结构，这也许会让你感到惊讶。谢天谢地，我们还可以用泛型Enumerable.OfType<T>()方法来对包含在这些非泛型集合里的数据进行迭代操作。

OfType<T>()方法没扩展泛型类型，它是Enumerable中少数几个这样的成员之一。当对一个实现了IEnumerable接口（例如ArrayList）的非泛型容器类调用这个成员方法时，只需指定容器中项的类型就可以提取一个兼容于IEnumerable<T>的对象。在代码中，可以使用隐式类型变量来存储该数据点。

考虑下面的新方法，它使用一组Car对象填充ArrayList（要在Program.cs文件中引入System.Collections命名空间）。

```
static void LINQOverArrayList()
{
  Console.WriteLine("***** LINQ over ArrayList *****");

  // 这是个非泛型的车集合
  ArrayList myCars = new ArrayList() {
    new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
    new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
    new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
    new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
    new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
  };

  // 把ArrayList转换成一个兼容于IEnumerable<T>的类型
  var myCarsEnum = myCars.OfType<Car>();

  // 建立兼容类型的查询表达式
  var fastCars = from c in myCarsEnum where c.Speed > 55 select c;

  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

与前面的示例类似，从Main()方法调用这个方法时，根据LINQ查询的格式，将只显示名字“Henry”和“Daisy”。

12.5.3 使用OfType<T>()筛选数据

你知道，非泛型类型可包含任何类型的项，因为这些容器类（再以ArrayList作例子）的成员的原型是接受System.Objects的。例如，假定一个ArrayList内包含好几个项，只有一些是数字。假如我们要得到只含数字类型数据的子集，可以使用OfType<T>()，因为它会过滤出那些类型不同于迭代操作中所指定类型的元素：

```
static void OfTypeAsFilter()
{
  // 从ArrayList中提取整数
  ArrayList myStuff = new ArrayList();
  myStuff.AddRange(new object[] { 10, 400, 8, false, new Car(), "string data" });
  var myInts = myStuff.OfType<int>();

  // 输出10、400和8
  foreach (int i in myInts)
  {
    Console.WriteLine("Int value: {0}", i);
  }
}
```

到目前为止，我们已经对数组、泛型集合和非泛型集合使用了LINQ查询。这些容器包含了基本类型（整型、字符串数据等）和自定义类。下面我们来学习其他的LINQ操作符，它们用于构建更加复杂和有用的查询。

源代码 LinqOverArrayList例程的源代码位于Chapter 12子目录下。

12.6 C# LINQ 查询操作符

C#定义了很多查询操作符，表12-3只定义了其中的几个操作符。

说明 .NET Framework 4.5 SDK文档提供了有关每一个C# LINQ操作符的细节。更多信息请参阅“LINQ General Progamming Guide”的主题。

表12-3 各种LINQ查询操作符

查询操作符	含义
from、in	用于定义任何LINQ表达式的主干，允许从合适的容器中提取数据子集
where	用于定义从一个容器里取出哪些项的限制条件
select	用于从容器中选择一个序列

（续）

查询操作符	含　义
join、on、equals、into	基于指定的键来做关联操作。记住，这些“关联”不必与关系数据库的数据有什么关系
orderby、ascending、descending	允许结果子集按升序或降序排序
group、by	用特定的值来对数据分组后得到一个子集

除了上面表12-3中列出的部分操作符外，System.Linq.Enumerable类还提供了一套没有直接的C#查询操作符简化符号，而是以扩展方法呈现的方法。可以调用这些泛型方法以各种方式来对一个结果集进行转换（Reverse<>()、ToArray<>()、ToList<>()等）。其中的一些方法用于从一个结果集提取单例，也有一些方法是对结果集进行集操作（Distinct<>()、Union<>()、Intersect<>()等），还有一些方法是对结果集进行聚合操作（Count<>()、Sum<>()、Min<>()、Max<>()等）。

```
class ProductInfo
{
  public string Name {get; set;}
  public string Description {get; set;}
  public int NumberInStock {get; set;}

  public override string ToString()
  {
    return string.Format("Name={0}, Description={1}, Number in Stock={2}",
      Name, Description, NumberInStock);
  }
}
```

现在在Main()方法中用ProductInfo对象填充数组：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Query Expressions *****\n");

  // 这个数组将是测试的基础
  ProductInfo[] itemsInStock = new[] {
    new ProductInfo{ Name = "Mac's Coffee",
                     Description = "Coffee with TEETH",
                     NumberInStock = 24},
    new ProductInfo{ Name = "Milk Maid Milk",
                     Description = "Milk cow's love",
                     NumberInStock = 100},
    new ProductInfo{ Name = "Pure Silk Tofu",
                     Description = "Bland as Possible",
                     NumberInStock = 120},
    new ProductInfo{ Name = "Cruchy Pops",
                     Description = "Cheezy, peppery goodness",
                     NumberInStock = 2},
    new ProductInfo{ Name = "RipOff Water",
                     Description = "From the tap to your wallet",
                     NumberInStock = 100},
    new ProductInfo{ Name = "Classic Valpo Pizza",
                     Description = "Everyone loves pizza!",
                     NumberInStock = 73}
  };
```

```
    // 这里我们将调用各种方法
    Console.ReadLine();
  }
```

12.6.1 基本的选择语法

因为LINQ查询表达式是在编译时校验的，记住这些操作符的次序是至关重要的。简单地说，每个LINQ查询表达式都是使用from、in和select操作符来建立的，如下所示：

```
var result = from matchingItem in container select matchingItem;
```

from操作符后面的项匹配LINQ查询条件，其名称可以任意选择。in操作符后面的项表示要查询的数据容器（数组、集合或XML文档）。

在此，我们的查询表达式不过是从容器里挑选出每个项而已，类似Select *SQL语句。考虑下面的代码：

```
static void SelectEverything(ProductInfo[] products)
{
  // 得到所有的对象
  Console.WriteLine("All product details:");
  var allProducts = from p in products select p;

  foreach (var prod in allProducts)
  {
    Console.WriteLine(prod.ToString());
  }
}
```

其实，这个查询表达式并不是很有用，因为我们的子集与传入的参数中的数据是一模一样的。如果我们愿意的话，可以使用传入的参数，通过使用下面的选择语法来只提取每个car对象的Name值。

```
static void ListProductNames(ProductInfo[] products)
{
  // 只提取产品的名字
  Console.WriteLine("Only product names:");
  var names = from p in products select p.Name;

  foreach (var n in names)
  {
    Console.WriteLine("Name: {0}", n);
  }
}
```

12

12.6.2 获取数据子集

为了从数据容器里得到特定的子集，可以使用where操作符。这么做时，总的语法模板成为下面这样：

```
var result = from item in container where BooleanExpression select item;
```

注意，where操作符预期一个运算结果是布尔值的表达式。例如，要从ProductInfo[]参数中提取库存量大于25的项，可以编写如下代码：

```
static void GetOverstock(ProductInfo[] products)
```

```
{
  Console.WriteLine("The overstock items!");

  // 只获取库存量大于25的项
  var overstock = from p in products where p.NumberInStock > 25 select p;

  foreach (ProductInfo c in overstock)
  {
    Console.WriteLine(c.ToString());
  }
}
```

本章前面介绍过，当构建一个where子句的时候，允许利用任意合法的C#操作符来构建复杂的表达式。例如，考虑下面这个只提取那些速度超过每小时100英里的BMW的查询：

```
// 得到速度最少是每小时100英里的BMW
var onlyFastBMWs = from c in myCars
                   where c.Make == "BMW" && c.Speed >= 100 select c;

foreach (Car c in onlyFastBMWs)
{
  Console.WriteLine("{0} is going {1} MPH", c.PetName, c.Speed);
}
```

12.6.3　投影新数据类型

从现有的数据源投影出新的数据形式也是可能的。让我们假定你想从传入的ProductInfo[]参数中得到一个只有产品的名字和描述的结果集。可以定义一个select语句，动态生成一个新的匿名类型：

```
static void GetNamesAndDescriptions(ProductInfo[] products)
{
  Console.WriteLine("Names and Descriptions:");
  var nameDesc = from p in products select new { p.Name, p.Description };

  foreach (var item in nameDesc)
  {
    // 也可以直接使用Name和Description属性
    Console.WriteLine(item.ToString());
  }
}
```

要记住在使用了投影的LINQ查询中，你无法知道实际的数据类型，因为它是在编译时决定的。在这种情况下，就必须使用var关键字。同样，你不能在创建方法时返回隐式类型，因此以下方法不能编译：

```
static var GetProjectedSubset(ProductInfo[] products)
{
    var nameDesc = from p in products select new { p.Name, p.Description };
    return nameDesc;  // 不行
}
```

在向调用者返回投影的数据时，可以使用ToArray()扩展方法将查询结果转换为.NET System. Array对象。因此，可以这样更新查询表达式：

```
// 现在返回Array
static Array GetProjectedSubset(ProductInfo[] products)
{
```

```
  var nameDesc = from p in products select new { p.Name, p.Description };

// 将匿名对象集映射为Array对象
  return nameDesc.ToArray();
}
```

在Main()中可以像下面这样调用和处理数据：

```
Array objs = GetProjectedSubset(ItemsInStock);
foreach (object o in objs)
{
  Console.WriteLine(o);  // 对每个匿名对象调用ToString()
}
```

注意，你需要使用字面的System.Array对象，而不能使用C#数组声明语法，因为你不知道编译器生成的匿名类的实际类型。同样，我们没有指定泛型方法ToArray<T>()的类型参数，因为直到编译时我们才能知道实际的数据类型，而这已经太晚了。

显然，由于Array对象中的项为Object类型，你无法继续享有强类型的优势。此外，在需要返回投影操作的LINQ结果集时，也必须将数据转换为Array类型（或通过Enumerable类型的其他成员转换为合适的容器）。

12.6.4 使用Enumerable获取总数

在投影一批新的数据时，你可能需要知道究竟有多少项返回到了序列中。如果要获取LINQ查询表达式返回的项数，可以简单地使用Enumerable类的扩展方法Count()。例如，下面的方法将查找本地数组中长度大于6个字符的string对象：

```
static void GetCountFromQuery()
{
  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
                                "Fallout 3", "Daxter", "System Shock 2"};

  // 获取查询出的总数
  int numb =
  (from g in currentVideoGames where g.Length > 6 select g).Count();

  // 打印项数
  Console.WriteLine("{0} items honor the LINQ query.", numb);
}
```

12.6.5 反转结果集

你可以简单地使用Enumerable类中的扩展方法Reverse<T>()对结果集中的项进行反转。例如，下面的方法对传入的ProductInfo[]参数的所有项进行反转：

```
static void ReverseEverything(ProductInfo[] products)
{
  Console.WriteLine("Product in reverse:");
  var allProducts = from p in products select p;
  foreach (var prod in allProducts.Reverse())
  {
    Console.WriteLine(prod.ToString());
  }
}
```

12.6.6 对表达式进行排序

在本章最开始的示例中，查询表达式使用orderby操作符通过一个指定的值对子集中的项进行排序。默认为正序。因此对字符串的排序为按字母顺序，对数字数据的排序为从小到大。如果要观察逆序排列的结果，可以使用descending操作符。考虑下面的方法：

```
static void AlphabetizeProductNames(ProductInfo[] products)
{
  // 按字母顺序获取产品的名称
  var subset = from p in products orderby p.Name select p;

  Console.WriteLine("Ordered by Name:");
  foreach (var p in subset)
  {
    Console.WriteLine(p.ToString());
  }
}
```

尽管默认为正序，你也可以使用ascending操作符，这样意图会更明确：

```
var subset = from p in products orderby p.Name ascending select p;
```

如果要使项逆序排列，就使用descending操作符：

```
var subset = from p in products orderby p.Name descending select p;
```

12.6.7 维恩图工具

Enumerable类提供了一些扩展方法，可以对两个（或多个）LINQ查询的数据进行合并（union）、比较（difference）、连接（concatenation）和交叉（intersection）。首先，考虑Except()扩展方法，它返回包含两个容器不同之处的LINQ结果集。在下例中，该值为“Yugo”：

```
static void DisplayDiff()
{
  List<string> myCars = new List<String> {"Yugo", "Aztec", "BMW"};
  List<string> yourCars = new List<String>{"BMW", "Saab", "Aztec" };

  var carDiff =(from c in myCars select c)
    .Except(from c2 in yourCars select c2);

  Console.WriteLine("Here is what you don't have, but I do:");
  foreach (string s in carDiff)
    Console.WriteLine(s);  // 打印Yugo
}
```

Intersect()方法返回两个容器中共同的数据项。例如，下面的方法返回的序列为“Aztec”和“BMW”。

```
static void DisplayIntersection()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

// 获得共同的成员
  var carIntersect = (from c in myCars select c)
```

```
    .Intersect(from c2 in yourCars select c2);

  Console.WriteLine("Here is what we have in common:");
  foreach (string s in carIntersect)
    Console.WriteLine(s); // 打印 Aztec和BMW
}
```

如你所想，Union()方法返回的是多个LINQ查询中的所有成员。与严格意义上的合并一样，如果相同的成员出现多次，将只能返回一个。因此，下面的方法将打印“Yugo”、“Aztec”、“BMW”和“Saab”：

```
static void DisplayUnion()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

  // 合并这两个容器
  var carUnion = (from c in myCars select c)
    .Union(from c2 in yourCars select c2);

  Console.WriteLine("Here is everything:");
  foreach (string s in carUnion)
    Console.WriteLine(s); // 打印所有公共成员
}
```

最后，Concat()扩展方法将返回直接连接的LINQ结果集。例如，下面的方法将打印“Yugo”、“Aztec”、“BMW”、“BMW”、“Saab”和“Aztec”：

```
static void DisplayConcat()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

  var carConcat = (from c in myCars select c)
    .Concat(from c2 in yourCars select c2);

  // 打印 Yugo Aztec BMW BMW Saab Aztec
  foreach (string s in carConcat)
    Console.WriteLine(s);
}
```

12.6.8 移除重复

在调用Concat()扩展方法时，最终得到的很可能是冗余的结果。某些情况下这就是你想要的，但某些情况下你可能希望移除数据中的重复条目。你可以简单地调用Distinct()扩展方法，如下所示：

```
static void DisplayConcatNoDups()
{
  List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };

  var carConcat = (from c in myCars select c)
    .Concat(from c2 in yourCars select c2);

  // 打印 Yugo Aztec BMW Saab Aztec
```

```
    foreach (string s in carConcat.Distinct())
      Console.WriteLine(s);
  }
```

12.6.9 LINQ聚合操作

LINQ查询可以对结果集执行不同的聚合操作。Count()扩展方法就是聚合的一个示例。其他类似的还有获取平均值、最大值、最小值、总值的Average()、Max()、Min()、Sum()方法，它们都位于Enumerable类中。例如：

```
static void AggregateOps()
{
  double[] winterTemps = { 2.0, -21.3, 8, -4, 0, 8.2 };

  // 不同的聚合示例
  Console.WriteLine("Max temp: {0}",
    (from t in winterTemps select t).Max());

  Console.WriteLine("Min temp: {0}",
    (from t in winterTemps select t).Min());

  Console.WriteLine("Avarage temp: {0}",
    (from t in winterTemps select t).Average());

  Console.WriteLine("Sum of all temps: {0}",
    (from t in winterTemps select t).Sum());
}
```

这些示例可以使你充分地了解构建LINQ查询表达式的过程。还有一些操作符没有在本章介绍，我们将在本书其他LINQ技术相关的章节中提供进一步的示例。本章最后将深入介绍C# LINQ查询操作符和实际对象模型的细节。

源代码 FunWithLinqExpressions项目的源代码位于Chapter 12子目录下。

12.7 LINQ 查询语句的内部表示

至此，你对使用各种C#查询操作符（例如from、in、where、orderby和select）来建立查询表达式的概念也有了初步的了解。同时，你还了解了LINQ to Object API的一些功能只能通过调用Enumerable类中的扩展方法来访问。然而实际上，在编译时，C#编译器将所有C# LINQ操作符都翻译为对Enumerable类中方法的调用。如果你不怕麻烦，完全可以只用实际的对象模型来构建LINQ语句。

Enumerable的许多方法的原型都是把委托（delegate）作为参数。特别是很多方法都要求一个名为Func<>的泛型委托（在第9章将泛型委托时介绍过）作为参数。例如，当使用C# where LINQ查询操作符时，将调用Enumerable的Where()方法：

```
// Enumerable.Where<T>()方法的重载版本
// 注意第二个参数的类型为System.Func<>
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
     System.Func<TSource,int,bool> predicate)
```

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
     System.Func<TSource,bool> predicate)
```

这个Fun<>委托（顾名思义）代表了一个接受一串（最多16个）参数和一个返回值的给定函数的模式。假如你用Visual Studio 对象浏览器检查这个类型的话，会注意到Func<>委托有很多形式如下所示：

```
// Func<>委托的各种格式
public delegate TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)

public delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)

public delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)

public delegate TResult Func<T1,TResult>(T1 arg1)

public delegate TResult Func<TResult>()
```

鉴于System.Linq.Enumerable的许多成员都要求一个委托作为输入，在调用它们时，我们可以手工创建一个新的委托类型，编写所需的目标方法，并使用C#的匿名方法，或者也可以定义一个合适的Lambda表达式。不管你用什么方法，最终结果是完全一样的。

使用C# LINQ查询操作符来构建一个LINQ查询表达式是最简单的方法。若这是正确的，那么我们来分析一下这些可行的方法，来看一下C#查询操作符和基础的Enumerable类型之间的联系。

12.7.1 用查询操作符建立查询表达式（复习）

首先，创建一个名为LinqUsingEnumerable的新LINQ控制台应用程序。Program类将定义一系列的静态辅助方法（都将在Main()方法中被调用）来示范我们建立查询表达式的各种方式。

第1个方法QueryStringsWithOperators()是建立查询表达式最直截了当的方式，跟我们前面的LinqOverArray例子里的代码完全一样，如下所示：

```
static void QueryStringWithOperators()
{
  Console.WriteLine("***** Using Query Operators *****");

  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  var subset = from game in currentVideoGames
               where game.Contains(" ") orderby game select game;

  foreach (string s in subset)
    Console.WriteLine("Item: {0}", s);
}
```

使用C#查询操作符来建立查询表达式的明显好处是，既看不到Func<>委托，也不用考虑对Enumerable类型的调用，因为做此类翻译是C#编译器的任务。毋庸置疑的是，使用各种查询操作符（from、in、where、orderby等）来建立LINQ表达式是最常见和最直截了当的方式。

12.7.2 使用Enumerable类型和Lambda表达式来建立查询表达式

记住，这里使用的LINQ查询操作符只不过是调用由Enumerable类型定义的各种扩展方法的速记版

本而已。下面这个QueryStringsWithEnumerableAndLambdas()方法直接使用了Enumerable扩展方法来处理局部的字符串数组：

```
static void QueryStringsWithEnumerableAndLambdas()
{
  Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");

  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // 用通过Enumerable类型赋予Array的扩展方法建立查询表达式
  var subset = currentVideoGames.Where(game => game.Contains(" "))
    .OrderBy(game => game).Select(game => game);

  // 输出结果
  foreach (var game in subset)
    Console.WriteLine("Item: {0}", game);
  Console.WriteLine();
}
```

这里，我们对currentVideoGames字符串数组调用了Where()扩展方法。Array类可以通过Enumerable中的扩展方法实现该操作。Enumerable.Where()方法要求System.Func<T1,TResult>委托类型的参数。该委托的第一个类型参数表示与IEnumerable<T>兼容的数据（本例中为字符串数组），第二个类型参数表示方法的结果数据，它是Lambda表达式中的单一语句所返回的。

在该代码示例中，Where()方法的返回值被隐藏了，但在后台我们操作的是OrderedEnumerable类型。对该类型调用泛型OrderBy()方法，该方法同样要求Func<>委托参数。这次，我们通过一个适合的Lambda表达式简单地依次传递各项。调用OrderBy()方法的最终结果是一个对原始数据重新排序的新序列。

最后，对OrderBy()返回的序列调用Select()方法，最终的数据集保存在隐式类型变量subset中。

的确，这种LINQ查询的“普通写法”要比之前的C#查询操作符示例复杂一些。毫无疑问，一部分复杂性是使用点操作符调用方法链造成的。下面的代码将查询拆分为多个语句，可以得到相同的结果：

```
static void QueryStringsWithEnumerableAndLambdas2()
{
  Console.WriteLine("***** Using Enumerable / Lambda Expressions *****");

  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // 拆分
  var gamesWithSpaces = currentVideoGames.Where(game => game.Contains(" "));
  var orderedGames = gamesWithSpaces.OrderBy(game => game);
  var subset = orderedGames.Select(game => game);

  foreach (var game in subset)
    Console.WriteLine("Item: {0}", game);
  Console.WriteLine();
}
```

你也许同意，与使用C#查询操作符相比，直接使用Enumerable类的方法来建立一个LINQ查询表达式比较烦琐。还有，Enumerable的方法要求委托作为参数，你需要经常编写Lambda表达式来提供将由底层委托目标处理的输入数据。

12.7.3 使用Enumerable类型和匿名方法来建立查询表达式

鉴于C# Lambda表达式只是用于操作匿名方法的简化符号，我们考虑一下QueryStringsWith-AnonymousMethods()辅助函数里定义的第3种查询表达式：

```
static void QueryStringsWithAnonymousMethods()
{
  Console.WriteLine("***** Using Anonymous Methods *****");

  string[] currentVideoGames = {"Morrowind", "Uncharted 2",
    "Fallout 3", "Daxter", "System Shock 2"};

  // 使用匿名方法建立所需的Func<>委托
  Func<string, bool> searchFilter =
    delegate(string game) { return game.Contains(" "); };
  Func<string, string> itemToProcess = delegate(string s) { return s; };

  // 把委托传递给Enumerable的方法
  var subset = currentVideoGames.Where(searchFilter)
    .OrderBy(itemToProcess).Select(itemToProcess);

  // 输出结果
  foreach (var game in subset)
    Console.WriteLine("Item: {0}", game);
  Console.WriteLine();
}
```

这个查询表达式的做法更加烦琐，因为我们手工编写了Enumerable类的Where()方法、OrderBy()方法和Select()方法所使用的Func<>委托。但从好的方面来看，匿名方法的句法确实把所有的委托处理都包含在单一方法定义中了。然而，这个方法在功能上与前面的QueryStringsWithEnumerable-AndLambdas()方法和QueryStringsWithOperators()方法完全等同。

12.7.4 用Enumerable类型和原始委托建立查询表达式

最后，如果我们真要使用非常烦琐的方式来建立一个查询表达式，可以避免使用Lambda或匿名方法句法，而是直接创建每个Func<>类型的委托目标。这是最后一次介绍查询表达式，我们在一个名为VeryComplexQueryExpression的新类中为其建模，如下所示：

```
class VeryComplexQueryExpression
{
  public static void QueryStringsWithRawDelegates()
  {
    Console.WriteLine("***** Using Raw Delegates *****");

    string[] currentVideoGames = {"Morrowind", "Uncharted 2",
      "Fallout 3", "Daxter", "System Shock 2"};

    // 建立所需的Func<>委托
    Func<string, bool> searchFilter = new Func<string, bool>(Filter);
    Func<string, string> itemToProcess = new Func<string,string>(ProcessItem);

    // 把委托传递给Enumerable的方法
    var subset = currentVideoGames
```

```
        .Where(searchFilter).OrderBy(itemToProcess).Select(itemToProcess);

      // 输出结果
      foreach (var game in subset)
        Console.WriteLine("Item: {0}", game);
      Console.WriteLine();
    }

    // 委托目标
    public static bool Filter(string game) {return game.Contains(" ");}
    public static string ProcessItem(string game) { return game; }
  }
```

可以按如下所示在Program类的Main()方法中调用这个方法来测试字符串处理的迭代：

```
VeryComplexQueryExpression.QueryStringsWithRawDelegates();
```

如果现在运行程序来测试各个可能的做法，你就不应该奇怪，不管走了哪条路，它们的输出都是一样的。请将下面关于LINQ查询表达式在底层如何表示的要点记在心里。

- 查询表达式是用各种C#查询操作符建立的。
- 查询操作符只是调用由System.Linq.Enumerable类型定义的扩展方法的简化符号。
- Enumerable的许多方法要求委托（特别是Func<>）作为参数。
- 任何要求委托参数的方法都可以传入一个Lambda表达式。
- Lambda表达式是伪装的匿名方法（这提高了可读性）。
- 匿名方法是指派一个原始委托，然后手工建立一个委托目标方法的简化符号。

这也许比你想要知道的更深入了一点，但我希望这个讨论能帮你理解C#查询操作符背后的工作原理。

说明　LinqUsingEnumerable项目的源代码位于Chapter 12子目录下。

12.8　小结

LINQ是一系列相关的技术，试图提供一个单一的、对称的方式来与各种形式的数据交互。就像本章所解释的那样，LINQ可以与任何实现了IEnumerable<T>接口的类型交互，这些类型包括简单数组、泛型的和非泛型数据集合。

你已经看到，LINQ技术是通过使用几个新的C#语言特性实现的。例如，由于LINQ查询表达式可以返回任意数目的结果集，使用var这个关键字来代表底层数据类型是很常见的。此外，Lambda表达式、对象初始化语法以及匿名类型都可用来建立函数式的、紧凑的LINQ查询。

更重要的是，C# LINQ查询操作符只是System.Linq.Enumerable类型静态方法调用的快捷形式。我们看到，大多数Enumerable的成员都操作Func<T>委托类型，它可以接受方法地址、匿名方法或者Lambda表达式作为输入来计算查询。[①]

---

① 进一步深入学习LINQ，推荐阅读《LINQ实战》（人民邮电出版社，2009）一书。——编者注

第 13 章

对象的生命周期

前一章讲述了怎样用C#创建自定义类类型。本章将介绍CLR怎样通过垃圾回收（garbage collection）来管理已分配的类实例（又称为对象）。C#程序员从来不直接从内存中删除一个托管对象（回忆一下，在C#语言中没有delete关键字）。相反，.NET对象被分配到一块叫做托管堆（managed heap）的内存区域上，在那里它们会在“将来的某一时刻”被垃圾回收器自动销毁。

了解了回收过程的核心细节之后，你将会明白怎样使用System.GC类类型通过编程使用垃圾回收器（在大多数.NET项目中你都不必这么做）。接着我们将分析怎样用System.Object.Finalize()虚方法和IDisposable接口建立类，而这些类能够可预测地、及时地释放内部非托管资源。

我们还将深入介绍.NET 4.0引入的一些关于垃圾回收器的新功能，包括后台垃圾回收和使用System.Lazy<>泛型类实现的延迟实例化。通过本章，你将会牢固地掌握CLR是怎样管理.NET对象的。

13.1　类、对象和引用

为了说清本章要研究的主题，进一步阐明类、对象和引用变量之间的不同是很重要的。类只是一个蓝图，它描述了这个类型的实例在内存中看起来是什么样子。当然，类是定义在一个代码文件中的（按习惯，在C#中通常带*.cs扩展名）。看看下面定义在名为SimpleGC的新控制台应用程序项目中的简单类Car：

```
// Car.cs
public class Car
{
  public int CurrentSpeed {get; set;}
  public string PetName {get; set;}

  public Car(){}
  public Car(string name, int speed)
  {
    PetName = name;
    CurrentSpeed = speed;
  }
  public override string ToString()
  {
    return string.Format("{0} is going {1} MPH",
      PetName, CurrentSpeed);
  }
}
```

定义了一个类后，就可以使用C#的new关键字分配任意数量的对象。但是，要理解new关键字返回的是一个指向堆上对象的引用，而不是真正的对象本身。如果在方法作用域中将引用变量声明为本地变量，这个引用变量保存在栈内，以供应用程序以后使用。当想调用对象中的成员时，可以使用点操作符：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** GC Basics *****");

    // 在托管堆上创建一个新的Car对象。返回了一个对这个对象的引用（'refToMyCar'）
    Car refToMyCar = new Car("Zippy", 50);

    // 用C#的点操作符（.）来调用使用了引用变量的对象的成员
    Console.WriteLine(refToMyCar.ToString());
    Console.ReadLine();
  }
}
```

图13-1说明了类、对象和引用的关系。

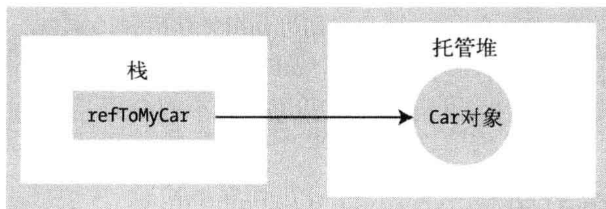


图13-1　对托管堆上对象的引用

说明　第4章介绍过，结构是值类型，它直接分配在栈上，而从来不会放在.NET托管堆上。只有在创建类的实例时，才会产生堆的分配。

13.2　对象生命周期的基础

当创建C#应用程序时，尽可以放心，你无需对托管堆进行直接操作，它将自动管理。事实上，在.NET上我们进行内存管理的规则非常简单。

规则　使用new关键字将一个类实例分配在托管堆上，然后就不用再管。

实例化结束后，垃圾回收器将会在对象不再需要时将其销毁。当然，紧接着就出现另一个问题了：垃圾回收器如何判断一个对象什么时候不再需要呢？简短（也就是不完全）的回答是：只有在一个对象从代码库的任何部分都不可访问时，垃圾回收器就会从堆中删除它。假设已有一个分配局部Car对象的Program类：

```
static void MakeACar()
{
  // 如果myCar只引用Car对象，当方法返回的时候它可能会被销毁
  Car myCar = new Car();
}
```

注意，Car引用（myCar）直接在MakeACar()方法中创建，并没有被传到该方法的定义作用域之外（通过返回值或ref/out参数）。因此，这个方法调用结束后，myCar引用就再也不访问，相关联的Car对象现在就是垃圾回收的候选目标。然而，要理解一点：不能保证这个对象会在MakeACar()完成后立即从内存中回收。但可以肯定的是，当CLR进行下一次垃圾回收时，myCar对象将被安全地销毁。

你肯定会发现，在有垃圾回收的环境中编程，将大大地简化应用程序的开发。这和C++中的编程形成鲜明的对比，在那里如果手工删除分配在堆上的对象失败，内存泄漏[①]就是迟早的事。事实上，跟踪寻找内存泄露是用非托管语言编程最耗时间的方面之一。通过允许垃圾收集器负责销毁对象，内存管理的麻烦都交给CLR了。

13.2.1　CIL的new指令

当C#编译器遇到new关键字时，它会在方法的实现中加入一条CIL newobj指令。如果编译上面的示例代码并使用ildasm.exe查看生成的程序集，将在MakeACar()方法中发现下面的CIL语句：

```
.method private hidebysig static void MakeACar() cil managed
{
  // 代码尺寸为8(0x8)
  .maxstack  1
  .locals init ([0] class SimpleGC.Car myCar)
  IL_0000:  nop
  IL_0001:  newobj instance void SimpleGC.Car::.ctor()
  IL_0006:  stloc.0
  IL_0007:  ret
} // Program::MakeACar方法结束
```

在研究有关对象何时从托管堆删除的严格规则之前，让我们更详细地探讨一下CIL的newobj指令的作用。首先，要理解托管堆不只是一个由CLR访问的随机内存块。.NET垃圾回收器是堆的“清洁工”，它会压缩空的内存块来实现优化（必要的时候）。为了辅助这一行为，托管堆保存着一个指针[②]（常称为下一个对象的指针或新对象指针），它精确地指示下一个对象将被分配的位置。

此外，newobj指令通知CLR执行下面的核心任务。

- 计算分配对象所需要的总内存数（包含数据成员和基类所需的内存）。
- 检查托管堆，确保有足够的空间来放置要分配的对象。如果空间足够，调用类型的构造函数，最终将内存中新对象的引用返回给调用者，它的地址恰好是下一个对象的指针的上一个位置。
- 最后，在将引用返回给调用者之前，移动下一个对象的指针，指向托管堆上的下一个可用的位置。

基本过程如图13-2所示。

① 未及时释放内存而导致内存无法再次使用，从而造成了浪费。——译者注

② pointer，是用于表示内存单元地址的变量。——译者注

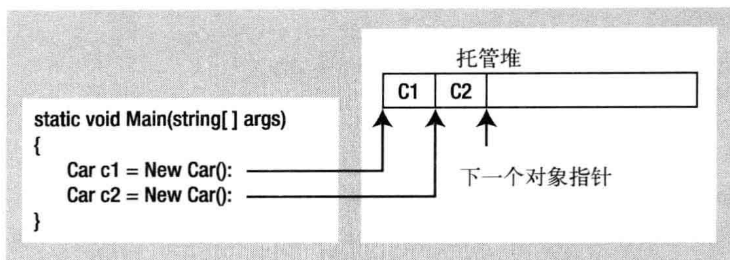


图13-2　在托管堆上分配对象的细节

因为我们经常需要在应用程序中频繁地分配对象，托管堆上的空间最终会用完。当处理newobj指令时，如果CLR判定托管堆没有足够的空间来分配所请求的类型，它会执行一次垃圾回收来尝试释放内存。因此，垃圾回收的下一个规则非常简单。

规则　如果托管堆没有足够的内存来分配所请求的对象，就会进行垃圾回收。

但垃圾回收具体是如何发生的，取决于应用程序正在使用的.NET平台的版本。本章将会介绍这些不同之处。

13.2.2　将对象引用设置为空

C/C++程序员通常将指针变量设置为null来确保不再引用非托管内存。因此，在C#中你可能会想知道，将对象引用赋值为null会怎么样。例如，假设MakeACar()子程序被更新为如下：

```
static void MakeACar()
{
  Car myCar = new Car();
  myCar = null;
}
```

如果我们将对象引用赋值为null，编译器会生成CIL代码来确保引用（这里是myCar）不再指向任何对象。如果我们再使用ildasm.exe来查看修改后的MakeACar()的CIL代码，将会发现ldnull操作码（在虚拟执行栈中产生一个null值），之后是stloc.o操作码（它用于设置null引用）：

```
.method private hidebysig static void  MakeACar() cil managed
{
  // 代码大小： 10(0xa)
  .maxstack  1
  .locals init ([0] class SimpleGC.Car myCar)
  IL_0000:  nop
  IL_0001:  newobj instance void SimpleGC.Car::.ctor()
  IL_0006:  stloc.0
  IL_0007:  ldnull
  IL_0008:  stloc.0
  IL_0009:  ret
} // Program::MakeACar方法结束
```

不管怎么样，我们必须知道，将引用赋值为null并不意味着强制垃圾回收器立即启动并把对象从

堆上移除。我们完成的唯一事情就是显式取消引用和之前引用所指向对象之间的连接。因此，和其他C系列的语言相比，在C#中将引用设置为null意义就不大了，不管怎么样，这样做也不会有什么害处。

13.3 应用程序根的作用

现在讨论垃圾回收器怎样确定什么时候“不再需要”一个对象。为了理解细节，你需要明白应用程序根（application root）的概念。简单地说，根（root）就是一个存储位置，其中保存着对托管堆上一个对象的引用。严格地说，根可以属于下面任何一个类别：

- 全局对象的引用（虽然在C#中不允许，但是CIL代码的确允许分配全局对象）；
- 静态对象/静态字段的引用；
- 应用程序代码库中局部对象的引用；
- 传递进一个方法的对象参数的引用；
- 等待被终结（finalize，本章后面将讲述）的对象的引用；
- 任何引用对象的CPU寄存器。

在一次垃圾回收过程中，运行库将检查托管堆上的对象，判断应用程序是否仍然可访问它们，即是否还是有根的（rooted）。为此，CLR将建立一个对象图，代表堆上可达的每一个对象。在我们讨论对象序列化时会再次看到对象图（见第20章）。现在，只要理解对象图是用来说明所有可访问对象的即可。还要明白，垃圾回收器从来不会在对象图中让同一个对象出现两次，这样就避免了COM编程中令人讨厌的循环引用计数（circular reference count）。

假设托管堆包含一个由A、B、C、D、E、F和G对象组成的集合。在一次垃圾回收中，检查这些对象（和它们可能包含的任何内部对象引用）是否有活动根。一旦构造好图，不可访问的对象（我们假设是对象C和F）就被标记为垃圾。图13-3就是对应于上述情景的一个可能的对象图（带方向的箭头表示依赖或需求，例如，“E依赖G，并间接依赖B”，“A不依赖任何对象”，等等）。

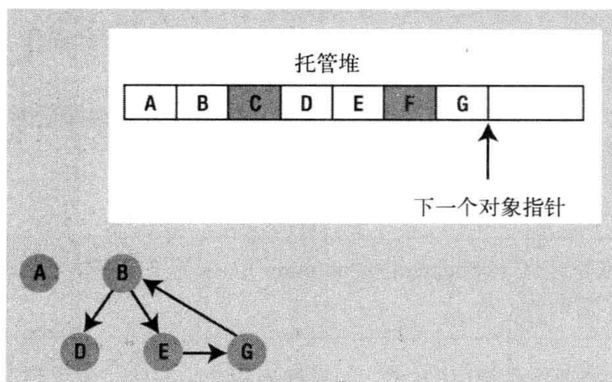


图13-3 建立对象图来确定哪些对象不能由应用程序根访问

一个对象被标记为终结后（这里是C和F，因为在对象图中没有表示它们），它们就会从内存中清除。这时，堆上剩余的空间被压缩，继而引起CLR修改活动应用程序根的集合（和隐含的指针），指

向正确的内存位置（这些都是自动并且透明[①]地完成的）。最后，下一个对象的指针被重新调整以指向下一个可用的位置。图13-4说明了重新调整的结果。

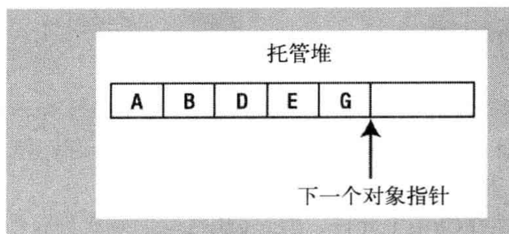


图13-4 一个无冗余并且压缩过的堆

说明 准确地说，垃圾回收器使用了两个不同的堆，一个专门用来存储非常大的对象。这个堆在回收周期中较少顾及，因为要重新定位[②]大对象的性能开销很大。尽管如此，认为“托管堆”是一个内存区域一般并没有什么问题。

13.4 对象的代

当CLR试图寻找不可访问的对象时，它不会逐个检查托管堆上的每一个对象。很明显，这样做将花费大量时间，尤其是在大一些（即实际）的应用程序中。

为了帮助优化这个过程，堆上的每一个对象被指定为属于某“代”（generation）。代的设计思路很简单：对象在堆上存在的时间越长，它就更可能应该保留。例如，定义桌面应用程序主窗口的类将一直停留在内存中直到程序结束。相反，最近才放在堆上的对象可能很快就不可访问了（例如在一个方法作用域中创建的对象）。基于这些假设，堆上的每一个对象都属于下列某代。

- 第0代：从没有被标记为回收的新分配的对象。
- 第1代：在上一次垃圾回收中没有被回收的对象（也就是，它被标记为回收，但因为已经获取了足够的堆空间而没有被删除）。
- 第2代：在一次以上的垃圾回收后仍然没有被回收的对象。

说明 第0代和第1代称为暂时代（ephemeral generation）。在下一节中你将看到，垃圾回收过程对于暂时代的处理是不同的。

垃圾回收器首先要调查所有的第0代对象。如果标记和清除这些对象得到了所需数量的空闲内存，任何没有被回收的对象都被提升到第1代。为了说明对象的代是怎样影响回收过程的，考虑图13-5，它

① transparent，用于修饰或说明一种进程或过程，它允许用户调用它，而用户并不需知道它的存在。——译者注

② relocate，移动计算机程序或它的一部分，并调整必要的基准地址，使得移动后的程序能够被（装入内存中）执行。——译者注

图示了一旦所需的内存被回收后，怎样提升没有被回收的第0代对象集合（A、B和E）。

如果算上所有的第0代对象后，仍然需要更多的内存，就会检查第1代对象的“可访问性”并相应地进行回收。没有被回收的第1代对象随后被提升到第2代。如果垃圾回收器仍然需要更多的内存，它会检查第2代对象的可访问性。这时，如果一个第2代对象在垃圾回收后仍然存在，它仍然是第2代对象，因为这是预定义的对象代的上限。

这里的要点是，通过给堆上的对象赋一个表示代的值，尽快地删除一些较新的对象（如本地变量），而不会经常“打扰”一些旧对象（例如程序的主窗体）。

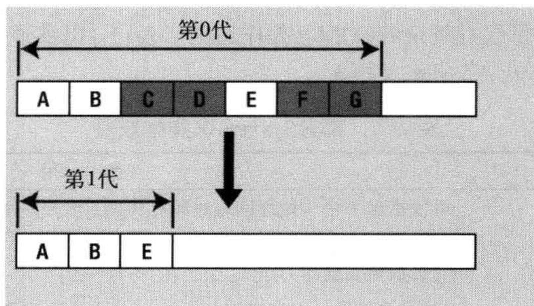


图13-5 在一次垃圾回收后幸存的第0代对象被提升到第1代

13.5 .NET 1.0 至.NET 3.5 的并发垃圾回收

在.NET 4.0之前，运行时使用*并发垃圾回收*技术来清理不再使用的对象。在这个模型下，当对第0代或第1代（前面说过这两代称为暂时代）对象执行回收时，垃圾收集器会暂时挂起当前进程中的所有活动线程，以确保应用程序在回收过程中不会访问托管堆。

我们将在第19章中介绍线程的知识，这里只需要简单地将其理解为正在运行的可执行文件中的执行路径。在垃圾回收周期完成时，挂起的线程可以继续工作。幸好.NET 3.5（及以前版本）的垃圾回收器经过高度优化，你很少能注意到应用程序中的这种短暂中断。

作为优化，并发垃圾回收通过专门的线程清理不在暂时代中的对象。这降低了（但并没有消除）判断活动线程时对.NET运行时的需求。此外，并发垃圾回收允许程序在回收非暂时代时继续分配堆上的对象。

13.6 .NET 4.0 及后续版本

从.NET 4.0开始，改变了垃圾回收器处理线程挂起的方式，它在清理托管堆上的对象时，使用*后台垃圾回收*。尽管它叫这个名称，但并不意味着所有的垃圾回收都发生在额外的后台执行线程。后台垃圾回收用于非暂时代上的对象，而对于暂时代上的对象，.NET运行时将使用一个专用后台线程进行回收。

.NET 4.0或更高版本的垃圾回收大大减少了一个给定线程参与垃圾回收细节时必须挂起的时间。这些改进的最终结果是，优化了清理第0代和第1代中无用对象的过程，并且提升了程序的运行时性能[这对那些需要较短（并且可预测的）GC中断时间的实时系统来说是非常重要的]。

但是，这种新的垃圾回收模式不会对如何构建.NET应用程序产生影响。实际上，.NET垃圾回收器可以不在人工直接干预的情况下执行其工作（微软也正在透明地改进回收过程）。

13.7　System.GC 类型

mscorlib.dll程序集提供了名为System.GC的类类型，它可以通过编程使用一些静态成员与垃圾回收器进行交互。这里要特别注意的是，极少需要在代码中直接使用这个类。一般情况下，只有在创建那些使用非托管资源的类时，才需要使用System.GC的成员。例如，使用.NET平台调用协议调用基于C的Windows API，或一些非常低级且复杂的COM互操作逻辑。表13-1提供了一些值得注意的成员（完整的细节请查询.NET Framework 4.5 SDK文档）。

表13-1　部分System.GC类型成员

System.GC的成员	作　用
AddMemoryPressure() RemoveMemoryPressure()	可以指定一个与垃圾回收过程相关的、代表调用对象"紧急性级别"（即压力）的数值。要明白这些方法应该一前一后地修改压力，这样就不会删除比所增加的总数更多的压力了
Collect()	强制GC进行一次垃圾回收。这个方法已被重载，以指定要回收的代和回收模式（通过GCCollectionMode枚举）
CollectionCount()	返回一个数值，表示已经对某一代进行了多少次垃圾回收
GetGeneration()	返回一个对象当前的代
GetTotalMemory()	返回当前分配在托管堆上的估计的内存数量（以字节为单位）。布尔参数指定调用是否在返回前等待垃圾回收的发生
MaxGeneration	返回目标系统支持的最大的代。在Microsoft的.NET 4.0下，可能有3代（0、1和2）
SuppressFinalize()	设置一个标志，说明这些对象不需要调用Finalize()方法
WaitForPendingFinalizers()	挂起当前线程直到所有可终结的对象都被终结。这个方法通常在调用GC.Collect()后被直接调用

为了了解如何使用System.GC类型来获取垃圾回收的细节，考虑下面的Main()方法，它使用了GC的部分成员：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with System.GC *****");

  // 输出堆上估计的字节数量
  Console.WriteLine("Estimated bytes on heap: {0}",
    GC.GetTotalMemory(false));

  // MaxGeneration是由0开始的，因此为了显示的目的加上了1
  Console.WriteLine("This OS has {0} object generations.\n",
    (GC.MaxGeneration + 1));

  Car refToMyCar = new Car("Zippy", 100);
  Console.WriteLine(refToMyCar.ToString());

  // 输出refToMyCar对象的代
```

```
    Console.WriteLine("Generation of refToMyCar is: {0}",
      GC.GetGeneration(refToMyCar));

    Console.ReadLine();
  }
```

强制垃圾回收

.NET垃圾回收器的功能是代替我们管理内存。然而，在一些非常罕见的环境下，通过编程使用GC.Collect()强制垃圾回收可能会有好处。以下是你可能会考虑与回收进程交互的两个常见场景：

- 应用程序将要进入一段代码，后者不希望被可能的垃圾回收中断；
- 应用程序刚刚分配非常多的对象，你想尽可能多地删除已获得的内存。

如果你确定让垃圾回收器马上检查不可访问对象可能有好处，就可以显式触发一次垃圾回收，如下所示：

```
static void Main(string[] args)
{
...
  // 强制一次垃圾回收，并等待每一个对象都被终结
  GC.Collect();
  GC.WaitForPendingFinalizers();
...
}
```

当手动强制垃圾回收时，应该总是调用GC.WaitForPendingFinalizers()。这样你可以稍等片刻，以确定在程序继续执行之前，所有可终结的对象都必须执行所有必要的清除工作。在底层，GC.WaitForPendingFinalizers()会在回收过程中挂起调用的“线程”。这是件好事，因为它保证代码不调用当前正在被销毁的对象的方法。

也可以给GC.Collect()方法提供一个数值，该数值表示垃圾回收将要操作的最老的代。例如，如果想让CLR只检查第0代对象，可以按如下方式编写代码：

```
static void Main(string[] args)
{
...
  // 只检查第0代对象
  GC.Collect(0);
  GC.WaitForPendingFinalizers();
...
}
```

还有，Collect()方法还可以传入GCCollectionMode枚举作为第二个参数，来调整运行库如何强制进行垃圾回收。enum定义了如下的值：

```
public enum GCCollectionMode
{
  Default,    // Forced是当前默认值
  Forced,     // 告诉运行库立即回收
  Optimized   // 允许运行库检测当前时间是否适合回收对象
}
```

像任何垃圾回收一样，调用GC.Collect()会提升没有被回收的对象的代。例如，按如下所示修改Main()方法：

13

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with System.GC *****");

  // 输出堆上估计的字节数
  Console.WriteLine("Estimated bytes on heap: {0}",
    GC.GetTotalMemory(false));

  // MaxGeneration是从0开始的
  Console.WriteLine("This OS has {0} object generations.\n",
    (GC.MaxGeneration + 1));
  Car refToMyCar = new Car("Zippy", 100);
  Console.WriteLine(refToMyCar.ToString());

  // 输出refToMyCar对象的代
  Console.WriteLine("\nGeneration of refToMyCar is: {0}",
    GC.GetGeneration(refToMyCar));

  // 为测试目的创建对象数组
  object[] tonsOfObjects = new object[50000];
  for (int i = 0; i < 50000; i++)
    tonsOfObjects[i] = new object();

  // 仅回收第0代对象
  GC.Collect(0, GCCollectionMode.Forced);
  GC.WaitForPendingFinalizers();

  // 输出refToMyCar对象的代
  Console.WriteLine("Generation of refToMyCar is: {0}",
    GC.GetGeneration(refToMyCar));

  // 看一下tonsOfObjects[9000]是否还活着
  if (tonsOfObjects[9000] != null)
  {
    Console.WriteLine("Generation of tonsOfObjects[9000] is: {0}",
      GC.GetGeneration(tonsOfObjects[9000]));
  }
  else
    Console.WriteLine("tonsOfObjects[9000] is no longer alive.");

  // 输出一个代被清除的次数
  Console.WriteLine("\nGen 0 has been swept {0} times",
    GC.CollectionCount(0));
  Console.WriteLine("Gen 1 has been swept {0} times",
    GC.CollectionCount(1));
  Console.WriteLine("Gen 2 has been swept {0} times",
    GC.CollectionCount(2));
  Console.ReadLine();
}
```

在这里，我们为了测试而创建了一个非常大的对象数组（精确数量是50 000）。从下面的输出结果中可以看到，即使这个Main()方法只做了一次垃圾回收的显式请求（通过GC.Collect()方法），CLR在幕后也执行了多次垃圾回收。

```
***** Fun with System.GC *****
Estimated bytes on heap: 70240
This OS has 3 object generations.
```

```
Zippy is going 100 MPH

Generation of refToMyCar is: 0
Generation of refToMyCar is: 1
Generation of tonsOfObjects[9000] is: 1

Gen 0 has been swept 1 times
Gen 1 has been swept 0 times
Gen 2 has been swept 0 times
```

本章讲到这里，希望你对对象生命周期的相关细节已经有了些认识。在下一节中，通过讲解怎样创建可终结（finalizable）对象和可处置（disposable）对象来更深入地分析垃圾回收过程。要明白一点：下面的技术仅在创建需要维护内部非托管资源的C#类时是非常必要的。

源代码 SimpleGC项目的源代码位于Chapter 13子目录下。

13.8 构建可终结对象

在第6章中，学习了.NET的基类System.Object，它定义了名为Finalize()的虚方法。毫无疑问，这个方法的默认实现是什么都不做的：

```
// System.Object
public class Object
{
  ...
  protected virtual void Finalize() {}
}
```

当为自定义的类重写Finalize()时，就建立了一个地方，来为类型执行必要的清理逻辑。因为这个成员被定义为受保护的，所以不可能通过点操作符从类实例中直接调用一个对象的Finalize()方法。相反，在从内存删除这个对象之前，垃圾回收器会调用对象的Finalize()方法（如果支持的话）。

说明 在结构类型上重写Finalize()是不合法的。这一点非常重要，因为结构是值类型，它们本来就从不分配在堆上，也就不能被回收。但是，如果你创建的结构包含需要清理的非托管资源，可以实现IDisposable接口（稍后将介绍）。

当然，Finalize()的调用将（最终）发生在一次“自然的”垃圾回收或可能用程序通过GC.Collect()强制回收的过程中。另外，当承载应用程序的应用程序域从内存中卸载时，会自动调用类型的终结器方法。有些读者可能知道应用程序域（或简写为AppDomain）是用来承载一个可执行程序集和任何需要的外部代码库的。如果对这个.NET概念不熟悉，不用着急，在学习完第17章后你自然会熟悉。简单地说，当应用程序域从内存中卸载时，CLR自动调用在它的生命周期中创建的每一个可终结对象的终结器。

现在，开发者的直觉告诉你，大多数C#类都不需要显式地清理逻辑，也不需要自定义终结器。原因很简单：如果类使用了其他托管对象，一切都最终会被回收。只是在你使用非托管资源时（例如原始的操作系统文件句柄、原始的非托管数据库连接、非托管内存或其他非托管资源），才可能需要设计一个在用完后清理自身的类。在.NET平台上，非托管资源是通过使用PInvoke（平台调用）服务直接调用操作系统的API，或通过一些复杂的COM交互获得的。因此，有垃圾回收的另一条规则。

规则　重写Finalize()的唯一令人信服的理由是，C#类通过PInvoke或复杂的COM互操作性任务使用了非托管资源（一般情况是通过System.Runtime.InteropServices.Marshal类型定义的各成员）。原因是在这些场景下，你无法操作没有托管在CLR的内存。

13.8.1　重写System.Object.Finalize()

在极其个别的情况下，确实需要创建一个使用非托管资源的类，你显然希望保证底层内存以一种可预测的方式被释放。假设创建了一个名为SimpleFinalize的新控制台应用程序项目并且插入一个名为MyResourceWrapper的使用非托管资源的类（无论怎样都是有可能的），且重写Finalize()。在C#中做这件事比较奇怪，不能用预期的override关键字来做：

```
class MyResourceWrapper
{
  // 编译时错误
  protected override void Finalize(){ }
}
```

当想配置自定义的C#类类型来重写Finalize()方法时，可以使用下面的（类似C++的）析构函数语法来达到同样的效果。之所以要用这种重写虚方法的替代形式，是因为当C#编译器执行一个终结器语法时，它将自动在被隐式重写的Finalize()方法中增加许多必需的基础代码（稍后介绍）。

C#终结器和构造函数很相似，因为它们和定义它们的类具有相同的名字。此外，终结器具有波浪号（~）前缀。然而，和构造方法不同的是，终结器不接受访问修饰符（它们是受隐式保护的），不接受参数，也不能被重载（一个类只能有一个终结器）。

下面是MyResourceWrapper自定义的终结器，它在被调用时会触发一声系统蜂鸣。很明显，这仅是为了教学的目的。一个实际的终结器应该只做清除任何非托管资源的工作，而不与其他托管对象的成员（即使是那些由当前对象引用的成员）进行交互，因为不能假设它们在垃圾回收器中调用Finalize()方法时还存在。

```
// 用终结器语法重写System.Object.Finalize()
class MyResourceWrapper
{
  ~MyResourceWrapper()
  {
    // 清除这里非托管的资源

    // 当被销毁时蜂鸣（仅为测试目的）
    Console.Beep();
  }
}
```

如果用ildasm.exe查看这个C#析构函数，将看到编译器加入了一些必要的错误检测代码。首先，Finalize()方法作用域内的代码段放在了一个try块中（见第7章）。相关的finally块保证基类的Finalize()方法总是被执行，而不管在try作用域内遇到的异常是什么。

```
.method family hidebysig virtual instance void
  Finalize() cil managed
{
  // 代码尺寸 13 (0xd)
  .maxstack  1
  .try
  {
    IL_0000:  ldc.i4     0x4e20
    IL_0005:  ldc.i4     0x3e8
    IL_000a:  call
    void [mscorlib]System.Console::Beep(int32, int32)
    IL_000f:  nop
    IL_0010:  nop
    IL_0011:  leave.s    IL_001b
  } // .try结束
  finally
  {
    IL_0013:  ldarg.0
    IL_0014:
      call instance void [mscorlib]System.Object::Finalize()
    IL_0019:  nop
    IL_001a:  endfinally
  } // 处理代码结束
  IL_001b:  nop
  IL_001c:  ret
} // MyResourceWrapper::Finalize方法结束
```

如果现在测试MyResourceWrapper类型，将发现系统蜂鸣发生在应用程序终止时，因为CIL会在AppDomain关闭时自动调用终结器。

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Finalizers *****\n");
  Console.WriteLine("Hit the return key to shut down this app");
  Console.WriteLine("and force the GC to invoke Finalize()");
  Console.WriteLine("for finalizable objects created in this AppDomain.");
  Console.ReadLine();
  MyResourceWrapper rw = new MyResourceWrapper();
}
```

13

源代码 SimpleFinalize项目的源代码位于Chapter 13子目录下。

13.8.2 终结过程的细节

始终要牢记，Finalize()方法的作用是保证.NET对象能在垃圾回收时清除非托管资源。这样，如果创建了一个不使用非托管内存的类型（到目前为止这是最常见的情况），终结是没有用的。事实上，只要有可能的话，就应该在设计类型时避免提供Finalize()方法，原因很简单，终结是要花费时间的。

当在托管堆上分配对象时，运行库自动确定该对象是否提供一个自定义的Finalize()方法。如果是这样，对象被标记为可终结的，同时一个指向这个对象的指针被保存在名为终结队列的内部队列中。终结队列是一个由垃圾回收器维护的表，它指向每一个在从堆上删除之前必须被终结的对象。

当垃圾回收器确定到了从内存中释放一个对象的时间时，它检查终结队列上的每一个项，并将对象从堆上复制到另一个称作终结可达表（finalization reachable table）（通常简写为freachable，读音为effreachable）的托管结构上。此时，下一个垃圾回收时将产生另一个线程，为每一个在可达表中的对象调用Finalize()方法。因此，为了真正终结一个对象，至少要进行两次垃圾回收。

总而言之，尽管对象的终结能够保证对象可以清除非托管的资源，但它本质上仍然是非确定的，而且由于额外的幕后处理，速度会变得相当慢。

13.9　构建可处置对象

当垃圾回收生效时，可以利用终结器来释放非托管资源。然而，因为很多非托管资源都非常宝贵（如数据库和文件句柄），所以它们应该尽可能快地被清除，而不能依靠垃圾回收的发生。除了重写Finalize()之外，类还可以实现IDisposable接口，它定义了一个名为Dispose()的方法：

```
public interface IDisposable
{
  void Dispose();
}
```

如果实现了IDisposable接口，就是假设当对象用户不再使用这个对象时，会在这个对象引用离开作用域之前手工调用Dispose()。这样，对象可以执行任何必要的非托管资源的清除工作，而且不会再有将对象放在终结队列上导致的性能损失，也不必等待垃圾回收器触发类的终结逻辑。

说明　结构和类类型都可以实现IDisposable（与重写Finalize()不同，后者只适用于类类型），因为对象用户（不是垃圾回收器）会调用Dispose()方法。

创建一个名为SimpleDispose的新控制台应用程序项目来说明IDisposable接口的使用。如下修改的MyResourceWrapper类现在实现了IDisposable，而没有重写System.Object.Finalize()：

```
// 实现IDisposable
class MyResourceWrapper : IDisposable
{
  // 对象用户应该在完成使用这个对象时调用这个方法
  public void Dispose()
  {
    // 在这里清除非托管资源

    // 抛弃包含的其他可处置对象

    // 出于测试目的
    Console.WriteLine("***** In Dispose! *****");
  }
}
```

注意，Dispose()方法不只负责释放一个对象的非托管资源，还应该对任何它包含的可处置对象调

用Dispose()。与Finalize()不一样，在Dispose()方法中与其他托管对象通信是很安全的。原因很简单：垃圾回收器并不支持IDisposable接口，永远都不会调用Dispose()。因此，当对象的用户调用这个方法时，对象仍然在托管堆上，并可以访问所有其他分配在堆上的对象。调用逻辑很直接：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Dispose *****\n");
    // 创建一个可处置对象并调用Dispose()，从而释放所有内部资源
    MyResourceWrapper rw = new MyResourceWrapper();
    rw.Dispose();
    Console.ReadLine();
  }
}
```

当然，在试图调用一个对象的Dispose()之前，需要保证类型支持IDisposable接口。虽然通过查询.NET Framwork 4.5 SDK文档，也可以知道哪一个基础类库类型实现了IDisposable，但可以使用在第6章讨论的is或as关键字通过编程来完成检查：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Dispose *****\n");
    MyResourceWrapper rw = new MyResourceWrapper();
    if (rw is IDisposable)
      rw.Dispose();
    Console.ReadLine();
  }
}
```

这个示例说明了内存管理的另一个规则。

规则 如果对象支持IDisposable，最好对任何直接创建的对象调用Dispose()。应该认为，如果类设计者选择支持Dispose()方法，这个类型就需要执行清除工作。如果忘记了这么做，内存仍将被清理（所以不必恐慌），只是会需要更长的时间。

对于之前的规则，还有一个补充：基础类库中的许多类型都实现了IDisposable接口，并且还提供了Dispose()方法的一个别名（有点混淆），这样使得定义类型的释放相关的方法听上去更自然。例如，System.IO.FileStream类实现了IDisposable（因此也支持Dispose()方法），它还定义了用于相同目的的Close()方法：

```
// 假设我们已经导入了System.IO命名空间
static void DisposeFileStream()
{
  FileStream fs = new FileStream("myFile.txt", FileMode.OpenOrCreate);

  // 确实有点混淆，这两个方法调用完成相同的事情
  fs.Close();
  fs.Dispose();
}
```

虽然关闭一个文件比释放更自然，但是你可能也觉得重复的清理的方法很容易混淆。由于提供别名的类型也不是很多，我们只要记住，如果类型实现了IDisposable，调用Dispose()总是安全的。

重用C#的using关键字

处理实现了IDisposable的托管对象时，经常需要使用结构化异常处理来保证类型的Dispose()方法在出现运行时异常时被调用：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");
  MyResourceWrapper rw = new MyResourceWrapper ();
  try
  {
    // 使用rw的成员
  }
  finally
  {
    // 无论是否发生错误，总是调用Dispose()
    rw.Dispose();
  }
}
```

虽然这是一个很好的防御性编程的例子，但实际情况是，如果只是为了保证调用Dispose()方法而把每个可处置的类型包装在try/finally块中，几乎不会有开发者愿意这么做。为了用更令人接受的方式达到同样的结果，C#提供了一个如下的特殊语法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");
  // 当退出using作用域时，自动调用Dispose()
  using(MyResourceWrapper rw = new MyResourceWrapper())
  {
    // 使用rw对象
  }
}
```

如果使用ildasm.exe查看Main()方法的CIL代码，会发现using语法确实用Dispose()调用扩展了try/finally逻辑：

```
.method private hidebysig static void Main(string[] args) cil managed
{
...
  .try
  {
    ...
  } // .try结束
  finally
  {
...
  IL_0012:  callvirt instance void
    SimpleFinalize.MyResourceWrapper::Dispose()
  } // 处理代码结束
...
} // Program::Main方法结束
```

说明　如果试图“使用”一个没有实现IDisposable的对象，将会收到编译器错误。

虽然这个语法的确免去了将可处置对象手工包装在try/finally逻辑中的需要，但C#的using关键字现在却有了双重含义（指定命名空间和调用Dispose()方法）。不过，当使用支持IDisposable接口的.NET类型时，这个句法结构将保证一旦退出了using块，“正在使用”的对象将自动调用其Dispose()方法。

同样，还要知道可以在using作用域中声明相同类型的多个对象。和我们期望的那样，编译器会插入代码来调用每一个声明对象的Dispose()。

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");

  // 使用逗号分割的列表来声明多个要释放的对象
  using(MyResourceWrapper rw = new MyResourceWrapper(),
                          rw2 = new MyResourceWrapper())
  {
    // 使用rw和rw2对象
  }
}
```

源代码　SimpleDispose项目的源代码位于Chapter 13子目录下。

13.10　构建可终结类型和可处置类型

现在，我们已看到了两种方式来构造能够清理内部非托管资源的类。一方面，我们可以使用终结器。这个技术我们尽可以放心使用，因为知道对象可以不需要用户参与进行垃圾回收，清除它自身。另一方面，我们可以实现IDisposable，给对象用户提供一种一旦对象用完就能清除的方法。然而，如果调用者忘记调用Dispose()，非托管资源可能会永远留在内存中。

将两个技术混合进同一个类定义是可行的。这样做可以获得两种模型的好处。如果对象用户记住了调用Dispose()，可以通过调用GC.SuppressFinalize()通知垃圾回收器跳过终结过程；如果对象用户忘记了调用Dispose()，对象最终也将被终结并有机会释放内部资源。对象的内部非托管资源会用其中一种方式释放掉。

下面是修改后的MyResourceWrapper，它现在既是可终结的，也是可处置的，它定义在C#控制台应用程序FinalizableDisposableClass中：

```
// 高级的资源包装器
public class MyResourceWrapper : IDisposable
{
  // 如果对象用户忘记调用Dispose()，垃圾回收器会调用这个方法
  ~MyResourceWrapper()
  {
    // 清除所有内部的非托管资源
    // 不要调用任何托管对象的Dispose()
  }
```

```
    // 对象用户将调用这个方法来尽快清除资源
    public void Dispose()
    {
      // 在这里清除非托管资源
      // 在其他包含的可处置对象上调用Dispose()

      // 如果用户调用了Dispose()就不需要终结，因此跳过终结
      GC.SuppressFinalize(this);
    }
  }
```

注意，Dispose()方法已被修改为调用GC.SupressFinalize()，它通知CLR在对象被回收时不再需要调用析构函数，因为非托管资源已经通过Dispose()逻辑被释放了。

正式的处置模式

上面的MyResourceWrapper实现的确能很好地工作，然而，它有一些小缺陷。首先，Finalize()和Dispose()方法都要清除相同的非托管资源。这当然导致了重复的代码，它很容易使维护复杂化。理想情况下，应该定义一个私有的辅助函数供两个方法调用。

下面，我们要确保Finalize()方法不会尝试处置任何托管对象，而Dispose()方法则应该这样做。最后，还要确保对象用户可以安全地多次调用Dispose()而不出错。当前，我们的Dispose()方法没有这样的安全措施。

为了解决这些设计问题，微软定义了一个正式甚至有些刻板的处置模式，它在健壮性、可维护性和性能三者间取得了平衡。下面是使用了这个官方模式的MyResourceWrapper的最终（且有注释的）版本：

```
class MyResourceWrapper : IDisposable
{
  // 用来判断Dispose()是否已经被调用
  private bool disposed = false;

  public void Dispose()
  {
    // 调用辅助方法
    // 指定true表示对象用户触发了清理过程
CleanUp(true);

    // 现在跳过终结
    GC.SuppressFinalize(this);
  }

  private void CleanUp(bool disposing)
  {
    // 保证我们还没有被处置
    if (!this.disposed)
    {
      // 如果disposing等于true，释放所有托管的资源
      if (disposing)
      {
        // 释放托管的资源
      }
      // 在这里清理非托管的资源
    }
```

```
      disposed = true;
    }

    ~MyResourceWrapper()
    {
      // 调用辅助方法
      // 指定false表示GC触发了清理过程
      CleanUp(false);
    }
  }
```

注意，MyResourceWrapper现在定义了名为CleanUp()的私有辅助方法。当指定true为参数时，表示对象用户初始化了清理过程，因此应该清理所有托管的和非托管的资源。然而，当垃圾回收器初始化清理过程时，调用CleanUp()时要指定false，以保证内部的可处置对象不被处置（因为我们不能假设它们仍然在内存中）。最后，布尔成员变量（disposed）在退出CleanUp()之前设置为true，以保证Dispose()可以被多次调用而不出错。

说明 当一个对象被“处置”后，它仍然在内存中，因此仍然可以在客户端调用其成员。因此，一个健壮的资源包装器类还需要更新类中的所有成员，添加这样的逻辑：“如果对象被处置了，就直接返回，什么都不做。”

为了测试MyResourceWrapper，在终结器作用域中加入对Console.Beep()的调用，如下所示：

```
~MyResourceWrapper()
{
  Console.Beep();
  // 调用辅助方法。指定false表示GC触发了清理过程
  CleanUp(false);
}
```

接下来修改Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Dispose() / Destructor Combo Platter *****");

  // 手动调用Dispose()，这不会调用终结器
  MyResourceWrapper rw = new MyResourceWrapper();
  rw.Dispose();

  // 不调用Dispose()，这会触发终结器，并引起嘟嘟声
  MyResourceWrapper rw2 = new MyResourceWrapper();
}
```

注意我们显式调用了rw对象的Dispose()方法，因此不会产生析构函数调用。然而，我们忘记调用rw2对象的Dispose()，因此应用程序终结时，我们会听到一声嘟嘟声。如果我们注释掉对rw对象调用Dispose()，就会听到两声嘟嘟声。

源代码 FinalizableDisposableClass项目的源代码位于Chapter 13子目录下。

我们对CLR通过垃圾回收器来管理对象的研究终于结束了。虽然垃圾回收过程还有一些深奥的细

节[例如弱引用（weak reference）和对象复苏（object resurrection）]没有分析，不过你通过前面的学习完全可以自己进行更深的探讨了。在本章最后，我们介绍一个.NET 4.0的新特性——对象的“延迟对象实例化”。

13.11　延迟对象实例化

在创建类时，可能偶尔会在代码中添加一个永远不会被使用的特殊成员变量，因为用户不会调用使用了它们的方法（或属性）。这很正常。但是，如果成员变量的初始化需要很大的内存空间，问题就来了。

例如，假设你正在编写一个封装了数字音乐播放器操作的类。除了预期的方法，如Play()、Pause()和Stop()之外，你还要返回一个Song对象集合（通过AllTracks类），表示设备上的各个数字音乐文件。

新建一个Console Application，命名为LazyObjectInstantiation，并定义如下的类类型：

```
// 表示一首歌曲
class Song
{
  public string Artist { get; set; }
  public string TrackName { get; set; }
  public double TrackLength { get; set; }
}

// 表示播放器上的所有歌曲
class AllTracks
{
  // 该多媒体播放器最多可容纳10 000首歌曲
  private Song[] allSongs = new Song[10000];

  public AllTracks()
  {
    // 假设用Song对象填充了数组
    Console.WriteLine("Filling up the songs!");
  }
}

// MediaPlayer包含AllTracks对象
class MediaPlayer
{
  // 假设这些方法包含相应的功能
  public void Play() { /* 播放歌曲 */ }
  public void Pause() { /* 暂停歌曲 */ }
  public void Stop() { /* 停止播放 */ }

  private AllTracks allSongs = new AllTracks();

  public AllTracks GetAllTracks()
  {
    // 返回所有的歌曲
    return allSongs;
  }
}
```

在当前的MediaPlayer实现中，我们假设用户希望通过GetAllTracks()方法获取歌曲的列表。但如果用户不需要这个列表呢？在我们当前的实现中，AllTracks成员变量仍将被分配，并在内存中创建10 000个Song对象，如下所示：

```
static void Main(string[] args)
{
  // 这次调用不需要获取所有的歌曲，但却间接地创建了10 000个对象
  MediaPlayer myPlayer = new MediaPlayer();
  myPlayer.Play();

  Console.ReadLine();
}
```

你显然不愿意创建10 000个没人使用的对象，因为这大大增加了.NET垃圾回收器的压力。尽管你可以手工添加一些代码来确保只有在使用的时候才创建allSongs对象（如使用工厂方法设计模式），但实际上还存在一种更简单的方式。

基础类库提供了一个非常有用的泛型类Lazy<>，它定义在mscorlib.dll内的System命名空间下。该类所定义的数据在代码库实际使用它之前是不会被创建的。由于它是一个泛型类，因此第一次使用时必须指定要创建的项的类型，可以是任意.NET基础类库中的类型或自定义类型。要让AllTracks成员变量支持延迟实例化，可以将下面的代码：

```
// MediaPlayer包含AllTracks对象
class MediaPlayer
{
...
  private AllTracks allSongs = new AllTracks();

  public AllTracks GetAllTracks()
  {
    // 返回所有歌曲
    return allSongs;
  }
}
```

改成：

```
// MediaPlayer包含Lazy<AllTracks>对象
class MediaPlayer
{
...
  private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();

  public AllTracks GetAllTracks()
  {
    // 返回所有歌曲
    return allSongs.Value;
  }
}
```

现在我们用Lazy<>类型来表示AllTracks成员变量，除此之外，还要注意GetAllTracks()方法也被修改了。在获取实际存储的数据时（本例中为包含10 000个Song对象的AllTracks对象），我们必须使用Lazy<>类的只读属性Value。

进行如此简单的修改之后，Main()方法只有在调用GetAllTracks()方法后才会间接分配Song对象，如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Lazy Instantiation *****\n");

  // 这里没有分配AllTracks对象
```

```
    MediaPlayer myPlayer = new MediaPlayer();
    myPlayer.Play();

    // 在调用GetAllTracks()时分配AllTracks
    MediaPlayer yourPlayer = new MediaPlayer();
    AllTracks yourMusic = yourPlayer.GetAllTracks();

    Console.ReadLine();
  }
```

说明　延迟对象初始化不仅可以降低不必要的对象分配，还可用于拥有昂贵创建代码的成员，如调用远程方法、连接关系型数据库，等等。

定制延迟数据的创建

在声明Lazy<>变量时，将使用默认的构造函数创建实际的内部数据类型：

```
// 在使用Lazy<变量>时将调用AllTracks的默认构造函数
private Lazy<AllTracks> allSongs = new Lazy<AllTracks>();
```

虽然这在某些情况下没有问题，但如果AllTracks类包含其他的构造函数，并且你想调用它们，会如何呢？此外，如果在建立Lazy<>变量时还有额外的工作要做（不只是创建AllTracks对象），这时又会如何呢？幸好Lazy<>类允许指定一个泛型委托作为可选参数。在创建其包装的类型时，它将调用指定的方法。

该泛型委托的类型为System.Func<>，它所指向的方法的返回值类型与相关的Lazy<>变量所创建的类型是相同的，并且可以包含16个参数（用泛型类型参数表示）。在大多数情况下，都不需要向Func<>所指向的方法传入任何参数。此外，为了简化Func<>的用法，我建议你使用Lambda表达式（委托与Lambda的关系详见第10章）。

记住MediaPlayer的最终版本在创建包装的AllTracks对象时添加了一些自定义代码。该方法在退出之前必须返回Lazy<>所包装对象的新实例，在创建该实例时可以选择任意的构造函数（这里我们仍然调用AllTracks的默认构造函数）：

```
class MediaPlayer
{
...
  // 在创建AllTracks对象时使用Lambda表达式添加额外的代码
  private Lazy<AllTracks> allSongs = new Lazy<AllTracks>( () =>
     {
       Console.WriteLine("Creating AllTracks object!");
       return new AllTracks();
     }
  );

  public AllTracks GetAllTracks()
  {
    // 返回所有的歌曲
    return allSongs.Value;
  }
}
```

希望你能看到Lazy<>类的用法。从根本上说，这个新的泛型类可以确保昂贵的对象只在用户需要的时候才进行分配。如果你觉得该话题对你的项目有用，可以查看.NET Framework 4.5 SDK文档中的System.Lazy<>类，来获取关于“延迟初始化”编程的更多示例。

源代码 LazyObjectInstantiation项目的源代码位于Chapter 13子目录下。

13.12 小结

本章的中心是解密垃圾回收过程。可以看到，垃圾回收器仅当它不能从托管堆获得所需要的内存时才运行（或是当指定的应用程序域从内存卸载时）。在有垃圾回收时，我们大可以放心，因为微软的回收算法已经通过使用对象代、用于对象终结的辅助线程和专门承载大对象的托管堆优化过了。

本章还举例说明了怎样通过程序使用System.GC类类型与垃圾回收器进行交互。前面提到，唯一真正需要这么做的情况是，创建在非托管资源上操作的可终结或可处置类类型。

回想一下，可终结类型是重写了System.Object.Finalize()虚方法以在回收垃圾时清除非托管资源的类。而可处置对象是提供了析构函数（通过重写Finalize()方法）类，对象用户将在IDisposable接口被实现后调用它。最后我们学习了结合两种方式的官方的“处置”模式。

本章结尾我们学习了.NET 4.0中的一个新泛型类Lazy<>。如你所见，使用该类可以将昂贵（也就是消耗内存大的）对象的创建推迟到用户实际需要的时候。这样，可以减少在托管堆中存储的对象数量，同时还能确保昂贵的对象只在真正需要的时候才由调用者创建。

13

Part 5 第五部分

用 .NET 程序集编程

本部分内容

第14章 .NET程序集入门

在本书前四部分中，我们学习创建了大量“独立”的可执行应用程序，所有的程序逻辑都打包在一个单独的可执行文件（*.exe）中。这些可执行程序集只是用了主要的.NET类库mscorlib.dll。尽管一些简单的.NET程序可以只使用.NET基类库构建，但对你（或你的同事）来说，将可复用的程序逻辑隔离到自定义的类库（*.dll文件）中供应用程序共享，会是司空见惯的事。

本章会学习将类型打包到自定义代码库中的多种方式。首先，我们会学习将类型隔离到命名空间的细节。之后，将介绍Visual Studio中的类库项目模板，并学习私有程序集和共享程序集的区别。

接下来我们将分析.NET运行库如何解析一个程序集的位置，并进而理解全局程序集缓存（GAC）、XML应用程序配置文件（*.config文件）、发行者策略程序集以及System.Configuration命名空间。

14.1 定义自定义命名空间

在深入库部署和配置的内容之前，首先要学习的是将自定义类型打包到.NET命名空间的细节。至此，我们已经构建了很多小的测试程序，它们使用.NET世界中既有的命名空间（特别是System）。然而，当构建包含很多类型的大型应用程序时，把相关类型分组到自定义命名空间中很有用。在C#中，这是通过namespace关键字来完成的。创建.NET *.dll程序集时，显式定义自定义命名空间就更重要了，因为其他开发者需要引用该库并导入我们的自定义命名空间来使用我们的类型。

为了直接研究这个问题，先创建一个新控制台应用程序CustomNamespaces。现在，假设我们在开发一组几何类Square、Circle和Hexagon。由于这种相似性，你可能希望把它们一起分组到CustomNamespaces.exe程序集MyShapes的唯一命名空间中。我们有两个基本方案。首先，我们可以选择把它们定义在一个C#文件（ShapesLib.cs）中，如下所示：

```
// ShapesLib.cs
using System;

namespace MyShapes
{
  // Circle类
  public class Circle { /* Interesting members... */ }

  // Hexagon类
  public class Hexagon { /* More interesting members... */ }

  // Square类
  public class Square { /* Even more interesting members... */ }
}
```

尽管C#编译器也允许在一个C#代码文件中包含多个类型，但如果你想在新项目中复用类定义就很麻烦了。例如，假设我们构建了一个新项目并只想使用Circle类。如果所有类型都定义在一个单独的代码文件中，你将或多或少受困于这个集合。因此，作为一种选择，你可以把一个命名空间分割到多个C#文件中。为了保证每个类型都封装在同一个逻辑组中，只需要把某个类定义封装在相同命名空间中即可，如下所示：

```
// Circle.cs
using System;

namespace MyShapes
{
  // Circle类
  public class Circle { /* Interesting methods... */ }
}

// Hexagon.cs
using System;

namespace MyShapes
{
  // Hexagon类
  public class Hexagon { /* More interesting methods... */ }
}

// Square.cs
using System;

namespace MyShapes
{
  // Square类
  public class Square { /* Even more interesting methods... */ }
}
```

注意，MyShapes命名空间成为这些类概念上的“容器”。如果另外一个命名空间（如CustomNamespaces）希望使用不同命名空间中的对象，可以使用using关键字，就像在.NET基础类库中使用命名空间时一样：

```
// 定义在基类库中的命名空间
using System;
// 使用定义在MyShapes命名空间中的类型
using MyShapes;

namespace CustomNamespaces
{
  public class Program
  {
    static void Main(string[] args)
    {
      Hexagon h = new Hexagon();
      Circle c = new Circle();
      Square s = new Square();
    }
  }
}
```

在这个示例中，我们假设定义MyShapes命名空间的C#文件与定义CustomNamespaces命名空间的文件都在同一个Console Application项目内。也就是说，所有的文件都将被编译为一个.NET可执行程序

集。如果在外部程序集中定义MyShapes命名空间，那么就必须引用那个库才能编译成功。你将在本章中学习使用外部库建立应用程序的所有细节。

14.1.1　使用完全限定名解决命名冲突

从技术角度说，当声明定义在外部命名空间内的类型时，不需要使用C#的using关键字。我们可以使用类型的完全限定名，回忆一下第1章，在类型名之前加上定义命名空间的前缀。例如：

```
// 注意，不再需要导入MyShapes了
using System;

namespace CustomNamespaces
{
  public class Program
  {
    static void Main(string[] args)
    {
      MyShapes.Hexagon h = new MyShapes.Hexagon();
      MyShapes.Circle c = new MyShapes.Circle();
      MyShapes.Square s = new MyShapes.Square();
    }
  }
}
```

一般没有必要使用完全限定名。不仅因为它需要更多的键盘敲击，还因为它在代码大小和执行时间上没有什么区别。其实，在CIL代码中，类型总是以完全限定名进行定义。因此，C#的using关键字只是节省了打字时间。

然而，完全限定名可以在使用包含同名类型的多个命名空间时避免命名冲突，此时就会很有用（可能也是必需的）。假设我们有一个新命名空间My3DShapes，它定义的3个类将图形以漂亮的3D进行呈现：

```
// 另一个Shape命名空间
using System;

namespace My3DShapes
{
  // 3D Circle类
  public class Circle { }

  // 3D Hexagon类
  public class Hexagon { }

  // 3D Square类
  public class Square { }
}
```

如果现在更新Program类，会遇到许多编译时错误，因为这些命名空间都定义了相同名字的类：

```
// 产生歧义
using System;
using MyShapes;
using My3DShapes;

namespace CustomNamespaces
{
  public class Program
  {
```

```
      static void Main(string[] args)
      {
        // 究竟引用哪个命名空间
        Hexagon h = new Hexagon(); // 编译器错误
        Circle c = new Circle();   // 编译器错误
        Square s = new Square();   // 编译器错误
      }
    }
  }
```

我们可以使用类型的完全限定名来解决歧义，如下所示：

```
// 现在就可以解决歧义了
static void Main(string[] args)
{
  My3DShapes.Hexagon h = new My3DShapes.Hexagon();
  My3DShapes.Circle c = new My3DShapes.Circle();
  MyShapes.Square s = new MyShapes.Square();
}
```

14.1.2 使用别名解决命名冲突

C# using关键字还可以用于创建类型完全限定名的别名。此时，我们就可以定义一个标识在编译时替代类型的完全限定名。例如，定义别名可以提供解决命名冲突的另一种方式：

```
using System;
using MyShapes;
using My3DShapes;

// 使用自定义别名来解决歧义
using The3DHexagon = My3DShapes.Hexagon;

namespace CustomNamespaces
{
  class Program
  {
    static void Main(string[] args)
    {
      // 这是在创建My3DShapes.Hexagon类
      The3DHexagon h2 = new The3DHexagon();
...
    }
  }
}
```

这种using语法还可以用来为很长的命名空间创建别名。在基础类库中有一个很长的命名空间是System.Runtime.Serialization.Formatters.Binary，它包含一个叫BinaryFormatter的成员。如果你愿意，可以按如下所示创建BinaryFormatter的实例：

```
using bfHome = System.Runtime.Serialization.Formatters.Binary;

namespace MyApp
{
  class ShapeTester
  {
    static void Main(string[] args)
    {
      bfHome.BinaryFormatter b = new bfHome.BinaryFormatter();
```

```
...

    }
  }
}
```

也可以使用传统的using指令：

```
using System.Runtime.Serialization.Formatters.Binary;

namespace MyApp
{
  class ShapeTester
  {
    static void Main(string[] args)
    {
      BinaryFormatter b = new BinaryFormatter();
        ...

    }
  }
}
```

在这里，不用关心BinaryFormatter类的用户（我们将在第20章中介绍该类）。目前，只需要简单地记住C# using关键字可以为较长的或常用的限定名定义别名，用来解决引入多个命名空间（它们定义相同的类型）时导致的名称冲突。

说明 滥用C#别名会导致代码库混乱。如果团队中的其他程序员不了解你的自定义别名，则很可能以为这些别名代表.NET基础类库中的类型，而当在.NET Framework 4.5 SDK文档中找不到这些标记时，这会带来相当大的困扰。

14.1.3 创建嵌套的命名空间

在组织类型的时候，完全可以在另一个命名空间中定义命名空间。.NET基础类库在许多地方这么用，用来提供类型组织的更深级别。例如，IO命名空间嵌套在System中以生成System.IO。如果希望创建一个根命名空间来包含既有的My3DShapes命名空间，可以按如下所示更新我们的代码：

```
// 嵌套一个命名空间
namespace Chapter14
{
  namespace My3DShapes
  {
    // 3D Circle类
    public class Circle{ }

    // 3D Hexagon类
    public class Hexagon{ }

    // 3D Square类
    public class Square{ }
  }
}
```

在很多情况下，根命名空间的作用只是提供更深级别的作用域，因此可能不会在其作用域中直接

定义任何类型（比如这里的Chapter14命名空间）。这样的话，可以使用如下的简洁形式来定义嵌套命名空间：

```
// 嵌套命名空间（组合两个）
namespace Chapter14.My3DShapes
{
  // 3D Circle类
  public class Circle{ }

  // 3D Hexagon类
  public class Hexagon{ }

  // 3D Square类
  public class Square{ }
}
```

由于我们在Chapter14根命名空间中嵌套了My3DShapes命名空间，所以就需要更新所有既有的using指令和类型别名：

```
using Chapter14.My3DShapes;
using The3DHexagon = Chapter14.My3DShapes.Hexagon;
```

14.1.4 Visual Studio 的默认命名空间

最后再提一下，在默认情况下，当使用Visual Studio 新建C#项目时，应用程序的默认命名空间名就是项目名。此后，如果我们使用Project→Add New Item菜单项来插入新代码文件，类型会自动使用默认命名空间进行包装。如果希望改变默认命名空间的名字，只需要使用项目Properties窗口的Application标签页来访问Default namespace（默认命名空间）选项（如图14-1所示）。

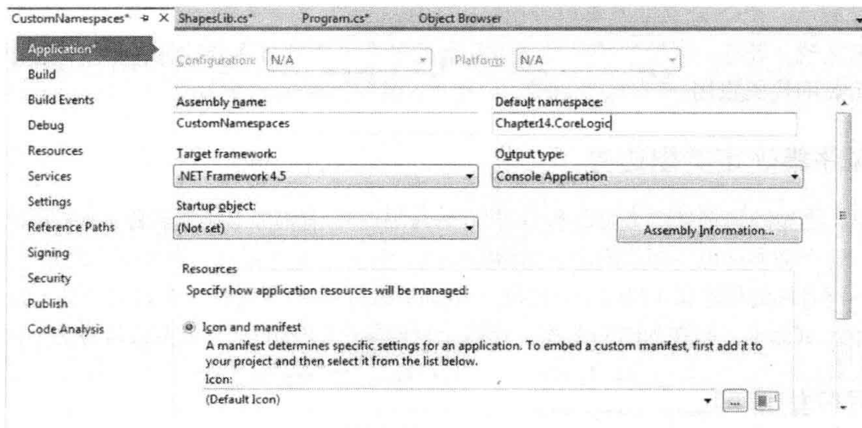


图14-1 配置默认命名空间

这样，任何插入到项目中的新项都会包装在Chapter14.CoreLogic命名空间中（显然，如果其他命名空间希望使用这些类型，就需要使用正确的using指令）。

现在，你已经了解了如何将自定义类型包装到组织良好的命名空间。接下来，让我们快速浏览.NET程序集的好处及其格式。然后，我们将深入介绍创建、部署和配置自定义类库的细节。

源代码　CustomNamespaces项目的源代码位于Chapter 14子目录下。

14.2　.NET 程序集的作用

.NET应用程序可以由多个程序集拼装而成。简单来说，程序集就是一个以公共语言运行库（CLR）为宿主的、版本化的、自描述的二进制文件。尽管现实中.NET程序集和以往Windows二进制文件的文件扩展名（*.exe或者*.dll）完全相同，但是两者的内部构成则几乎完全不同。因此，为了给后续内容做铺垫，这里先分析一下程序集的内部格式带来的好处。

14.2.1　程序集促进代码重用

从表面上看，前面章节中构建的控制台应用程序中所有的应用逻辑都包含在一个可执行程序集中。而实际上，它们所使用的许多类型来自于总是可访问的.NET代码库——mscorlib.dll（回忆一下前面章节，C#编译器会自动引用mscorlib.dll）以及System.Core.dll（在某些示例中）。

我们知道，代码库（也称类库）是一个*.dll形式的文件，它包含了一些外部应用程序能够调用的类型。当创建可执行程序集时，毫无疑问，我们将需要使用许多系统提供的和自己手工编写的代码库。但是，请记住一点：代码库的文件扩展名不一定是*.dll。一个可执行程序集完全可以使用定义在外部可执行文件中的类型。因此，一个被引用的*.exe也可以被认为是代码库。

无论一个代码库如何被打包，.NET平台允许我们以语言无关的方式来重用其中的类型。例如，可以使用C#创建一个代码库，然后在其他.NET编程语言中重用它。不但可以在不同的语言间分配类型，而且可以相互继承。用C#定义的基类可以由Visual Basic编写的类来扩展，用F#定义的接口可以由C#定义的结构来实现，等等。这里的关键是：当需要把单个可执行文件分拆为多个.NET程序集时，就能够做到语言无关的代码重用。

14.2.2　程序集确定类型边界

回忆一下，类型的完全限定名由命名空间（如`System`）为前缀加上类型名称（如`Console`）组成。然而严格地说，这个类型标识（即前面说的类型完全限定名）中还需要加上类型所在的程序集。比方说，如果有两个不同名称的程序集（MyCars.dll和YourCars.dll），它们同时在一个命名空间（`CarLibrary`）中定义了一个`SportsCar`类，则在.NET世界里，这两个程序集中的`SportsCar`类型被认为是不同的。

14.2.3　程序集是可版本化的单元

每个.NET程序集被分配一个格式为<*major*>.<*minor*>.<*build*>.<*revision*>的四部分数字版本号。（如果没有显式设置版本号，程序集会自动分配版本号1.0.0.0，即默认的Visual Studio项目设置。）版本号加上可选的公钥值使得一个程序集的不同版本在同一台机器上能够共处而不产生冲突。正式来讲，提供了公钥信息的程序集被称为强命名的（strongly named）。在本章后面你将看到，通过使用强名称，CLR可以确保客户端调用程序能够加载正确版本的程序集。

14.2.4 程序集是自描述的

程序集被认为是自描述的（self-describing），部分原因是它们记录了自身正常运行所需要访问的外部程序集。因此，如果程序集需要访问System.Windows.Forms.dll和System.Core.dll，那么它们的信息将会被记录到该程序集的清单（manifest）中。在第1章中曾经讲过，清单是一段描述程序集自身各种信息（名称、版本、需要的外部程序集等）的元数据块。

除了清单数据，程序集还包含另外一些元数据。这些元数据描述了程序集包含的每一个类型的组成（成员名称、实现的接口、基类、构造函数等）。由于程序集的信息被如此详细地记录下来，CLR并不需要访问Windows系统注册表来解析程序集的位置（这与微软原来的COM编程模型有很大的区别）。随着本章的阅读，你将会发现CLR使用一种完全新颖的方式去解析外部代码库的位置。

14.2.5 程序集是可配置的

程序集可以以“私有”或者“共享”的方式部署。私有程序集与调用它的客户端应用程序处于同一个目录下（也可以在其子目录下）。而共享程序集则是为了让同一台机器上的大部分应用程序都可以使用它而存在的一种代码库，它被部署在*全局程序集缓存*（GAC）的特定目录中。

无论怎样部署程序集，都可以编写基于XML的配置文件。使用这些配置文件，可以指示CLR做到：在指定位置查找程序集，为特定客户端加载指定版本的程序集，查阅本地机器、网络位置或基于Web的URL上的任意目录。阅读完本章，我们将会对XML配置文件有更好的理解。

14.3 .NET 程序集的格式

我们已经知道了.NET程序集所带来的好处，现在进一步深入程序集的内部结构。从结构上看，一个.NET程序集（*.dll或者*.exe）包含以下几个部分：

- Windows文件首部
- CLR文件首部
- CIL代码
- 类型元数据
- 程序集清单
- 可选的嵌入资源

通常头两个部分（Windows文件首部和CLR文件首部）容易被忽视，但其实它们是值得简要了解一下的。下面来逐个讲解。

14.3.1 Windows文件首部

Windows文件首部使程序集可以被Windows系列操作系统加载和操作。这些首部信息标识了应用程序将以什么类型（是基于控制台、基于图形用户界面还是*.dll代码库）驻留于Windows操作系统中。使用dumpbin.exe工具（通过Developer Command Prompt），打开一个.NET程序集，并通过`dumpbin/headers CarLibrary.dll`指定`/headers`标记，我们可以浏览该程序集的Windows文件首部信息。

下面是本章稍后要创建的CarLibrary.dll程序集的（部分）Windows首部信息（如果你想立即运行dumpbin.exe，可以用本书编写的任意*.dll或*.exe的名称来替换CarLibrary.dll）：

```
Dump of file CarLibrary.dll

PE signature found
File Type: DLL

FILE HEADER VALUES
             14C machine (x86)
               3 number of sections
        4B37DCD8 time date stamp Sun Dec 27 16:16:56 2011
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
            2102 characteristics
                   Executable
                   32 bit word machine
                   DLL

OPTIONAL HEADER VALUES
             10B magic # (PE32)
            8.00 linker version
             E00 size of code
             600 size of initialized data
               0 size of uninitialized data
            2CDE entry point (00402CDE)
            2000 base of code
            4000 base of data
          400000 image base (00400000 to 00407FFF)
            2000 section alignment
             200 file alignment
            4.00 operating system version
            0.00 image version
            4.00 subsystem version
               0 Win32 version
            8000 size of image
             200 size of headers
               0 checksum
               3 subsystem (Windows CUI)
...
```

记住，大多数的.NET程序员都没必要关心内嵌在.NET程序集中的首部数据的格式。除非碰巧要构建一个新的.NET语言编译器（这时你才会在乎这些信息），否则你肯定对不用去理会首部数据中那些晦涩的细节感到十分高兴。但要注意的是，当Windows向内存中加载二进制图像时，在后台将使用这些信息。

14.3.2 CLR文件首部

为了驻留于CLR中，所有的.NET文件都必须含有CLR首部数据块。简单地讲，CLR文件首部定义了多个标记，它们使得运行库可以了解到托管文件的布局。例如，这些标记标识了文件中元数据和资源的位置、程序集构建的运行库版本、（可选的）公钥值等。使用代码`dumpbin/clrheader`

CarLibrary.dll为dumpbin.exe工具提供/clrheader标记，我们就可以浏览该程序集内部的CLR首部信息，如下所示：

```
Dump of file CarLibrary.dll

File Type: DLL

  clr Header:
              48 cb
            2.05 runtime version
            2164 [     A74] RVA [size] of MetaData Directory
               1 flags
                   IL Only
               0 entry point token
               0 [       0] RVA [size] of Resources Directory
               0 [       0] RVA [size] of StrongNameSignature Directory
               0 [       0] RVA [size] of CodeManagerTable Directory
               0 [       0] RVA [size] of VTableFixups Directory
               0 [       0] RVA [size] of ExportAddressTableJumps Directory
               0 [       0] RVA [size] of ManagedNativeHeader Directory

  Summary
        2000 .reloc
        2000 .rsrc
        2000 .text
```

同样，.NET开发人员不需要过多关心程序集的CLR头部信息细节。只需要理解每一个.NET程序集都包含这个数据，它在背后被.NET运行库作为图片数据加载进内存。现在我们看一下每天的编程任务中非常有用的信息。

14.3.3 CIL代码、类型元数据和程序集清单

程序集的核心部分包含CIL代码，这些CIL代码是独立于平台和CPU的中间语言。在运行时，程序集内部的CIL代码才被（实时的JIT编译器）编译成特定平台和CPU的指令。在这种机制下，.NET程序集可以在多种不同的架构、设备和操作系统下运行。（虽然可以完全不去了解CIL编程语言的细节，但是在第18章还是对其语法和语义进行了介绍。）

程序集还包含元数据，这些元数据完整地描述了程序集内含类型和引用外部类型的格式。.NET运行库利用元数据在内存的二进制布局类型中解析类型（以及类型的成员）的位置，使远程方法调用更便利。在第15章探讨反射服务的时候，将会详细讲述.NET元数据的格式。

另外，程序集必须被关联上一个清单（manifest，也称*程序集元数据*）。该清单详细记录了程序集中的每一个模块、构建程序集的版本以及该程序集引用的所有外部程序集。在本章中，我们将会看到CLR广泛地使用程序集的清单来定位外部程序集引用。

14

14.3.4 可选的程序集资源

最后，.NET程序集还可以包含一些嵌入资源，如应用程序图标、图像文件、声音片段或者字符串表。事实上，.NET平台支持*卫星程序集*（satellite assemblie），这些程序集只包含本地化资源。在构建

国际化软件系统的时候，我们可能想基于特定区域（英语、德语等）来对资源进行分类打包，这时候附属程序集就显得非常有用。附属程序集不是本书要讨论的话题，如果感兴趣的话，请参考.NET 4.5 Framework文档了解卫星程序集的有关信息。

14.4　构建和使用自定义类库

在开始进入.NET类库的世界前，首先创建一个包含几个公共类型的单文件*.dll程序集（名为CarLibrary）。使用Visual Studio可以这样构建一个代码库，只需通过File→New Project菜单项选择类库（Class Library）项目（如图14-2所示）。

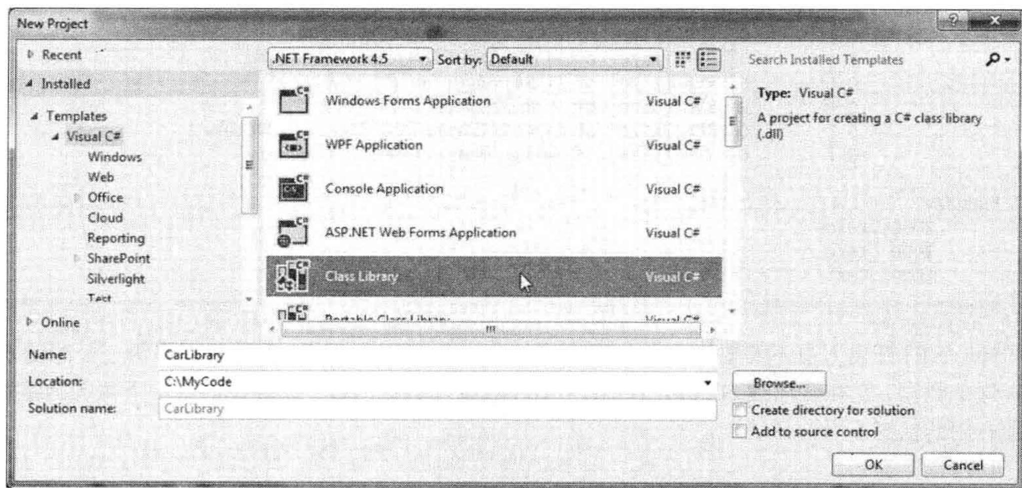


图14-2　创建一个C#类库

在这个汽车类库中，首先创建一个名为Car的抽象基类，它通过自定义属性语法定义了各种状态数据。它还拥有一个抽象方法TurboBoost()，使用自定义的枚举类型（EngineState）来表示汽车引擎的当前状态，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CarLibrary
{
  // 表示引擎的状态
  public enum EngineState
  { engineAlive, engineDead }

  // 层次结构中的抽象基类
  public abstract class Car
  {
    public string PetName {get; set;}
    public int CurrentSpeed {get; set;}
    public int MaxSpeed {get; set;}
```

```
    protected EngineState egnState = EngineState.engineAlive;
    public EngineState EngineState
    {
      get { return egnState; }
    }

    public abstract void TurboBoost();

    public Car(){}
    public Car(string name, int maxSp, int currSp)
    {
      PetName = name; MaxSpeed = maxSp; CurrentSpeed = currSp;
    }
  }
}
```

现在假设Car类型有两个子类：MiniVan（旅行车）和SportsCar（跑车）。每个子类都通过Windows Forms消息框显示恰当的消息来重写抽象方法TurboBoost()。在DerivedCars.cs项目中插入新的C#类文件，代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// 先读代码！只有添加了.NET类库才能通过编译
using System.Windows.Forms;

namespace CarLibrary
{
  public class SportsCar : Car
  {
    public SportsCar(){ }
    public SportsCar(string name, int maxSp, int currSp)
      : base (name, maxSp, currSp){ }

    public override void TurboBoost()
    {
      MessageBox.Show("Ramming speed!", "Faster is better...");
    }
  }

  public class MiniVan : Car
  {
    public MiniVan(){ }
    public MiniVan(string name, int maxSp, int currSp)
      : base (name, maxSp, currSp){ }

    public override void TurboBoost()
    {
      // Minivans引擎的马力不足
      egnState = EngineState.engineDead;
      MessageBox.Show("Eek!", "Your engine block exploded!");
    }
  }
}
```

14

请注意，每一个子类实现TurboBoost()时，都使用了Windows Forms的MessageBox类，该类定义在System.Windows. Forms.dll程序集中。为使程序集使用在外部程序集中定义的类型，因此需要通过Add Reference对话框来为CarLibrary项目添加对外部二进制文件的引用（如图14-3所示）。Add Reference对话框可以通过选择Visual Studio Project→Add Reference菜单项打开。

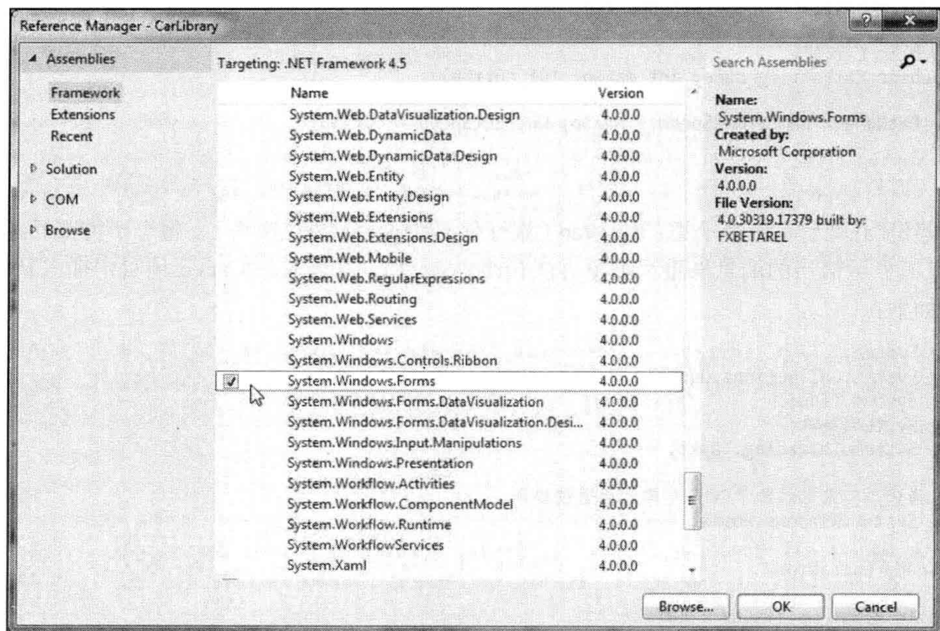


图14-3　使用Add Reference对话框引用外部.NET程序集

这里必须清楚的一点是：Add Reference对话框的Framework区域并没有显示机器上所有的程序集。Add Reference对话框并不会显示我们自定义的库，也不会显示所有部署在GAC（全局程序集缓存，本章后面会介绍其细节）中的库，它只显示Visual Studio预先设定的一组常用程序集。当构建的应用程序需要添加一个在Add Reference对话框中未列出的程序集时，我们需要单击Browse节点手动找到该*.dll或*.exe。

说明　Add Reference对话框的Recent节会不断更新显示最近被引用的程序集。这很方便，因为.NET项目通常使用同样的外部库核心设置。

14.4.1　清单

在客户端应用程序使用CarLibrary.dll之前，先查看一下代码库的内部结构。假定项目已经编译完毕，现在通过File→Open菜单，导航到CarLibrary项目的\bin\Debug子目录，将CarLibrary.dll加载到ildasm.exe中。这样，就可以在IL disassembler工具中查看你的库了（如图14-4所示）。

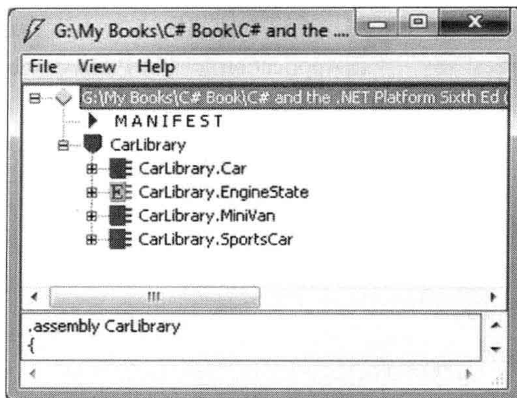


图14-4　将CarLibrary.dll加载到ildasm.exe中

双击MANIFEST图标，打开CarLibrary.dll的清单。该清单的第一个代码块描述了当前程序集正确运行所必需的所有外部程序集。你应记得，CarLibrary.dll使用了mscorlib.dll和System.Windows. Forms.dll里面的类型，因此两者都在清单中通过`.assembly extern`标记列出，如下所示：

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
```

每一个`.assembly extern`块由`.publickeytoken`指令和`.ver`指令组成。`.publickeytoken`指令仅当程序集被配置为**强名称**（在“强名称”部分会详细介绍）的时候才需要。而`.ver`指令则用于表示引用程序集的数字版本标识。

在这些外部引用信息之后，我们会发现很多`.custom`标记，它们标识了程序集级别的特性（版权信息、公司名、程序集版本等）。下面是manifest数据的部分代码：

```
.assembly CarLibrary
{
  .custom instance void ...AssemblyDescriptionAttribute...
  .custom instance void ...AssemblyConfigurationAttribute...
  .custom instance void ...RuntimeCompatibilityAttribute...
  .custom instance void ...TargetFrameworkAttribute...
  .custom instance void ...AssemblyTitleAttribute...
  .custom instance void ...AssemblyTrademarkAttribute...
  .custom instance void ...AssemblyCompanyAttribute...
  .custom instance void ...AssemblyProductAttribute...
  .custom instance void ...AssemblyCopyrightAttribute...
...
  .ver 1:0:0:0
}
.module CarLibrary.dll
```

通常，可以使用当前项目的Properties编辑器可视地建立这些设置。现在，回到Visual Studio，点击Solution Explorer中的Properties图标，单击Application选项卡中的Assembly Information...按钮（默认选中），将弹出如图14-5所示的GUI编辑器。

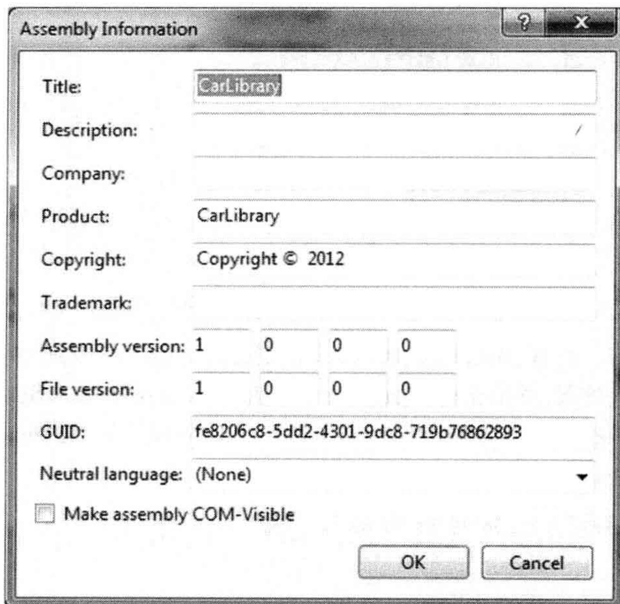


图14-5　使用Visual Studio的Properties编辑器编辑程序集信息

保存设置，GUI编辑器将更新项目中的AssemblyInfo.cs文件。该文件由Visual Studio维护，并且展开Solution Explorer中的Properties节点，可以看到它，如图14-6所示。

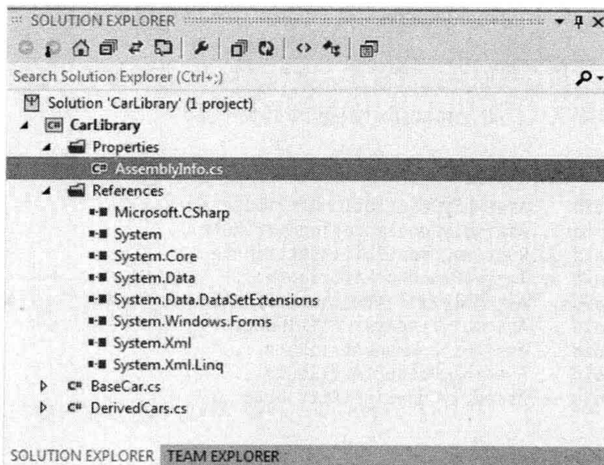


图14-6　使用GUI Properties编辑器可以更新AssemblyInfo.cs文件

浏览该文件，可以看到大量中括号括起的.NET特性，例如：

```
[assembly: AssemblyTitle("CarLibrary")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("CarLibrary")]
[assembly: AssemblyCopyright("Copyright © 2012")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

第15章会详细介绍这些特性，因此这里不用担心其细节。但必须知道的是，定义在AssemblyInfo.cs文件里面的大量特性会用来更新程序集清单中的.custom标记的值。

14.4.2 CIL

前面介绍过，程序集并不包含特定平台的指令。相反，它包含的是独立于平台的CIL指令。当.NET运行库把一个程序集加载进内存时，CIL将会被（JIT编译器）编译成目标平台可以理解的指令。如果回到ildasm.exe中，双击SportsCar类的TurboBoost()方法，ildasm.exe会打开一个新窗口展示实现该方法的CIL指令：

```
.method public hidebysig virtual instance void
  TurboBoost() cil managed
{
  // 代码大小18(0x12)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr  "Ramming speed!"
  IL_0006:  ldstr  "Faster is better..."
  IL_000b:  call  valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult
    [System.Windows.Forms]System.Windows.Forms.MessageBox::Show(string, string)
  IL_0010:  pop
  IL_0011:  ret
} // 方法SportsCar::TurboBoost结束
```

同样，尽管大多数.NET开发者不需要深入了解CIL的细节，但第18章还是详细介绍了它的语法和语义。信不信由你，理解CIL的语法将有助于构建更加复杂的需要高级服务的应用程序，如程序集的运行时构造（同样，详见第18章）。

14.4.3 类型元数据

在构建使用自定义.NET库的应用程序之前，如果按下Ctrl+M组合键，ildasm.exe会显示CarLibrary.dll程序集中的每一个类型的元数据（如图14-7所示）。

第15章会研究程序集的元数据，对.NET平台来说，它是非常重要的元素，并且是许多技术的脊柱（对象序列化、延迟绑定、可扩展应用程序等）。现在已经对CarLibrary.dll程序集的内部结构有一定的了解，接下来开始构建使用你的类型的客户端应用程序。

源代码 CarLibrary项目的源代码位于Chapter 14子目录下。

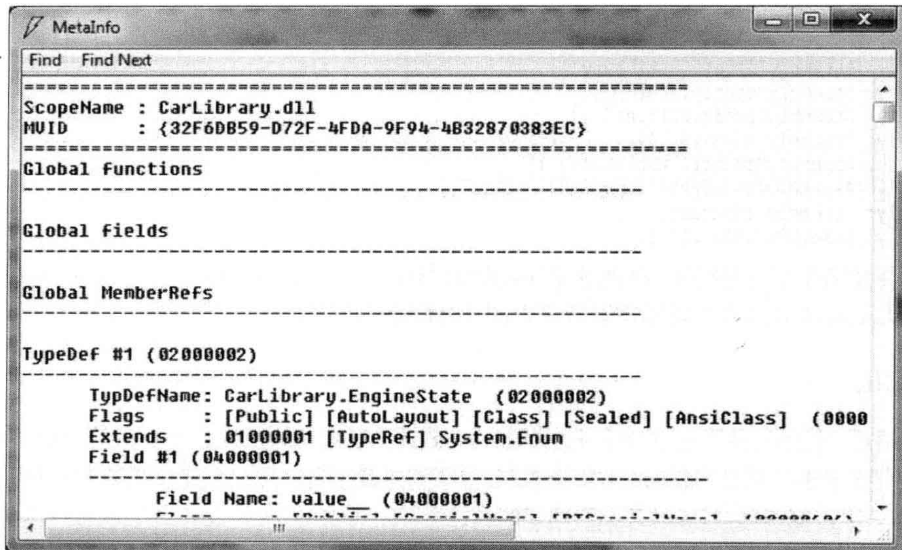


图14-7　CarLibrary.dll中的类型元数据

14.4.4　构建C#客户端应用程序

由于CarLibrary中的每一个类型都被声明为public，因此其他.NET应用程序都可以使用它们。其实可以使用internal关键字把类型定义为内部的（实际上，C#默认的访问方式是internal）。内部类型只能被它所在的程序集使用，而不能被外部客户端看到和创建。

为了能够调用库的功能，我们创建一个C#控制台应用程序项目CSharpCarClient，然后使用Add Reference对话框的Browse节点添加对CarLibrary.dll的引用（如果使用Visual Studio编译CarLibrary.dll，程序集将被放到CarLibrary项目文件夹的子目录\bin\Debug下）。现在，我们可以构建客户端应用程序去使用这些外部类型了。把先前的C#文件做如下更改：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

// 不要忘记导入命名空间CarLibrary
using CarLibrary;

namespace CSharpCarClient
{
  public class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** C# CarLibrary Client App *****");
      // 创建SportsCar对象
      SportsCar viper = new SportsCar("Viper", 240, 40);
```

```
            viper.TurboBoost();

            // 创建MiniVan对象
            MiniVan mv = new MiniVan();
            mv.TurboBoost();

              Console.WriteLine("Done. Press any key to terminate");
            Console.ReadLine();
        }
    }
}
```

以上代码跟先前编写的其他应用程序看上去没有什么区别。唯一的不同是这个C#客户端应用程序使用定义在另一个自定义库里面的类型。现在运行程序，它会打开多个不同的消息框。

你可能想知道在使用Add Reference对话框引用CarLibrary.dll时到底会发生什么。点击Solution Explorer中的Show All Files按钮，会发现Visual Studio 将原始的CarLibrary.dll复制到了CSharpCarClient项目文件夹的子目录\bin\Debug下（如图14-8所示）。

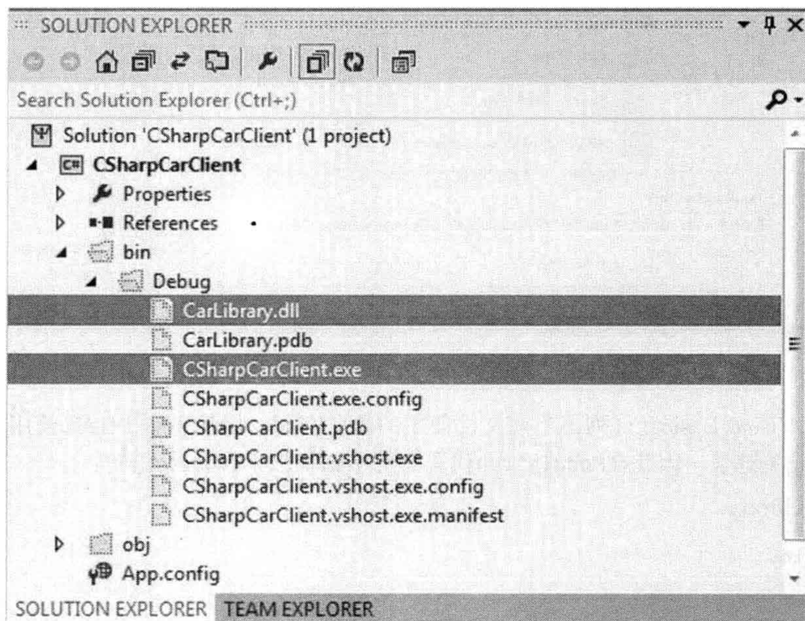


图14-8 Visual Studio 将私有程序集复制到客户目录中

本章后面将会介绍，CarLibrary.dll被部署为“私有”程序集（这是Visual Studio 类库项目的自动行为）。当你在新应用程序中引用私有程序集时（如CSharpCarClient.exe），IDE会通过在客户端应用程序的输出目录中放置库的副本来响应。

源代码 CSharpCarClient项目的源代码位于Chapter 14子目录下。

14.4.5　构建 Visual Basic 客户端应用程序

.NET平台允许开发者跨编程语言共享编译的代码。为了阐明.NET平台的语言无关性，让我们使用Visual Basic来构建新的控制台应用程序（VisualBasicCarClient），如图14-9所示。先创建一个项目，然后使用Add Reference对话框添加对CarLibrary.dll的引用。通过选择Project→Add Reference菜单选项来打开Add Reference对话框。

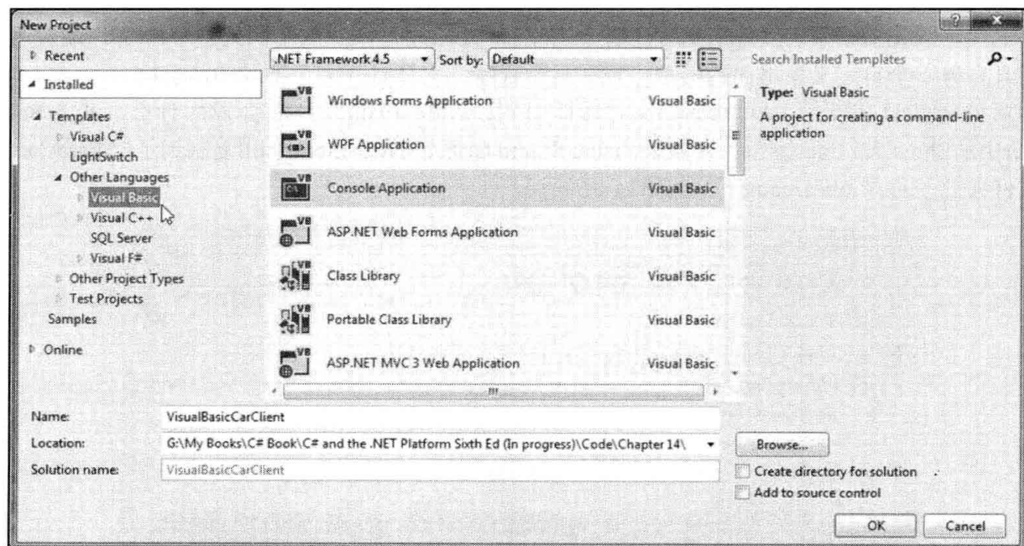


图14-9　创建Visual Basic控制台应用程序

与C#一样，Visual Basic允许列出当前文件使用的命名空间。但是Visual Basic使用Imports关键字而不是C#的using关键字，因此在Module1.vb代码文件中添加以下Imports语句：

```
Imports CarLibrary

Module Module1
  Sub Main()
  End Sub
End Module
```

注意，Main()方法定义在Visual Basic Module类型中。简而言之，Module类型是Visual Basic对只含有静态方法（与C#的静态很像）的类的简写方式。为了使用Visual Basic语法处理MiniVan和SportsCar类型，把Main()方法改写为：

```
Sub Main()
  Console.WriteLine("***** VB CarLibrary Client App *****")

  '通过Dim关键字声明本地变量
  Dim myMiniVan As New MiniVan()
  myMiniVan.TurboBoost()

  Dim mySportsCar As New SportsCar()
```

```
    mySportsCar.TurboBoost()
    Console.ReadLine()
  End Sub
```

编译后，运行本应用程序，将再次看到程序打开了一连串的对话框。另外，新的客户端应用程序将它自己的CarLibrary.dll本地副本放置在bin\Debug文件夹下。

14.4.6 实现跨语言继承

使用.NET开发最吸引人的一个方面就是跨语言继承。为了说明这一点，我们创建一个新的Visual Basic类，用它继承（之前使用C#编写的）SportsCar。首先，（通过选择Proiect→Add Class菜单项）为当前Visual Basic应用程序添加一个新的类文件，命名为PerformanceCar.vb。使用Inherits关键字更改该类的定义，让它继承SportsCar类型。然后，使用Overrides关键字重写抽象方法TurboBoost()，如下所示：

```
Imports CarLibrary

'这个VB类型继承了用C#编写的SportsCar类型
Public Class PerformanceCar
  Inherits SportsCar

  Public Overrides Sub TurboBoost()
    Console.WriteLine("Zero to 60 in a cool 4.8 seconds...")
  End Sub

End Class
```

为了测试新的类类型，更新模块的Main()方法，如下所示：

```
Sub Main()
...
  Dim dreamCar As New PerformanceCar()

  '使用继承的属性
  dreamCar.PetName = "Hank"
  dreamCar.TurboBoost()
  Console.ReadLine()
End Sub
```

注意，dreamCar对象可以调用继承链中存在的任何公共成员（如PetName属性），而不用担心基类完全由不同的语言定义并且位于完全不同的程序集中。以语言无关的方式跨程序集边界对类进行扩展，是.NET开发周期中一个非常自然的部分。你可以方便地使用编译后的由那些不愿用C#的人编写的代码。

14

源代码 VisualBasicNetCarClient项目的源代码位于Chapter 14子目录下。

14.5 私有程序集

到现在为止，我们在本章中所创建的类库都被部署为私有程序集。私有程序集要求放置在客户端

应用程序所在目录（应用程序目录）或者其子目录下。在先前构建CSharpCarClient.exe和VisualBasicCarClient.exe的过程中，添加对CarLibrary.dll的引用的时候，Visual Studio会把CarLibrary.dll复制到客户端应用程序目录下（至少在首次编译结束后）。

当一个客户端程序使用定义在外部程序集的类型时，CLR只是加载CarLibrary.dll本地的副本。由于.NET运行库在查找被引用的程序集时并不查询系统注册表，因此我们可以把CSharpCarClient.exe（或者VisualBasicCarClient.exe）和CarLibrary.dll程序集一起放到机器上的某个位置，然后运行应用程序（这就是时常听到的Xcopy部署）。

卸载（复制）排他性使用私有程序集的应用程序非常容易，只需直接删除（复制）应用程序文件夹就可以了。更重要的是，不需要担心移除私有程序集会破坏机器上其他应用程序的正常运行。

14.5.1　私有程序集的标识

私有程序集的全标识包括友好名称和数字版本号，两者都被记录在程序集清单中。*友好名称*就是模块（包含程序集清单的）名字减去文件扩展名。例如，如果查看CarLibrary.dll程序集，会看到如下内容：

```
.assembly CarLibrary
{
...
  .ver 1:0:0:0
}
```

鉴于私有程序集的独立性，CLR在解析它的位置时并不忙于使用它的版本号。我们假定私有程序集并不需要详细检查版本，因为客户端应用程序是唯一知道其存在的实体。因此，一台机器上很有可能会有同一个私有程序集的副本在不同目录下多次出现的情况。

14.5.2　探测过程

.NET运行环境使用一种叫*探测*（probing）的技术解析私有程序集的位置，这项技术其实远不如它的名字听起来那样具有侵略性。探测是一种把外部程序集请求映射到被请求的二进制文件位置的过程。严格来说，一个加载请求可以是*显式的*或者*隐式的*。隐式的加载请求发生在CLR查询清单的.assembly extern标记来解析程序集位置的时候，例如：

```
// 隐式加载请求
.assembly extern CarLibrary
{ ... }
```

显式的加载请求则发生在以编程方式调用System.Reflection.Assembly类的Load()或者LoadFrom()方法时，这两个方法主要在后期绑定或者动态调用类型成员时用到。第15章有相关的详细介绍，以下就是一个显式加载请求的示例代码：

```
// 根据友好名称显式加载请求
Assembly asm = Assembly.Load("CarLibrary");
```

不管是隐式还是显式，在CLR获得程序集的友好名称之后，便开始探测客户端应用程序目录下的CarLibrary.dll文件。如果找不到文件，它就尝试去查找具有相同友好名称的可执行程序集（如CarLibrary.exe）。如果还是找不到，运行库就在运行时引发FileNotFoundException异常。

说明　准确地说，如果在客户端应用程序目录下找不到被请求的程序集副本，CLR会尝试查找该目录下具有程序集友好名称的子目录（例如C:\MyClient\CarLibrary）。如果被请求的程序集在这个子目录下，CLR就可以成功地把该程序集加载到内存中。

14.5.3 配置私有程序集

虽然可以采用这种部署方式——把所需的程序集复制到（用户硬盘中）单个文件夹，但是我们可能更想定义多个子目录来区分相关内容。例如，假设应用程序目录是C:\MyApp，其中包含CSharpCarClient.exe。在该目录下有一个名为MyLibraries的子目录，其中放有CarLibrary.dll。

不考虑两个目录的关系，只有提供一个配置文件，CLR才会探测到MyLibraries子目录。配置文件包含多个XML元素，它们可以决定探测的过程。根据“规则”，配置文件必须拥有和运行的应用程序相同的名称，带*.config的文件扩展名，同时，它们必须被部署到客户端应用程序目录下。因此，如果为CSharpCarClient.exe创建配置文件的话，该配置文件的名称必须是CSharpCarClient. exe.config，并且位于C:\MyApp目录下（就本例而言）。

现在让我们实践一下。首先使用Windows资源管理器在C盘创建一个名为MyApp的新目录，然后将CSharpCarClient.exe和CarLibrary.dll复制到这个目录，双击前者运行程序。你的程序应该可以成功运行。接着在C:\MyApp下创建一个名为MyLibraries的子目录（如图14-10所示），移动CarLibrary.dll到该目录中。

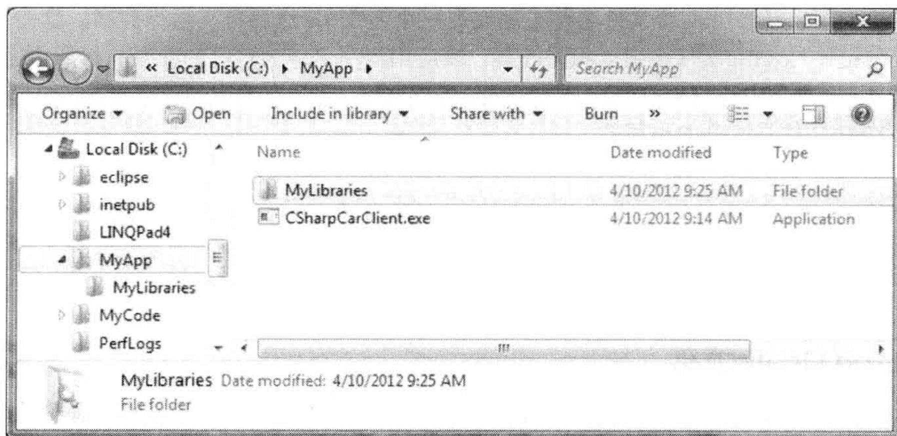


图14-10　现在CarLibrary.dll位于MyLibraries子目录下

再次双击该可执行文件尝试运行程序。由于CLR在应用程序目录下不能直接找到CarLibrary，因此你会收到一个令人厌恶的且未被处理的`FileNotFoundException`异常。

为了指示CLR在MyLibraries下进行探测，让我们创建一个名为CSharpCarClient.exe.config的配置文件，然后把它放到CSharpCarClient.exe所处的目录下（C:\MyApp）。打开该文件，输入以下内容（注意XML是区分大小写的）：

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="MyLibraries"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

.NET的*.config文件通常以<configuration>元素作为根元素。内嵌的<runtime>元素可能还定义了一个<assemblyBinding>子元素，该子元素会进一步内嵌一个<probing>元素。在这个例子里面，privatePath特性是关键，它用于通知CLR需要在应用程序目录下的哪些子目录下进行探测。

注意<probing>元素并没有指明哪一个程序集位于它指定的子目录下。换句话说，不能认为“CarLibrary就在MyLibraries子目录下，而MathLibrarie在OtherStuff子目录下”。<probing>元素只是指示CLR到所有指定的子目录去查找被请求的程序集，直到找到第一个匹配的。

说明 注意privatePath特性并不能用于指定一个绝对路径（C:\SomeFolder\SomeSubFolder）或者一个相对路径（..\SomeFolder\AnotherFolder）!如果想指定应用程序目录以外的另一个目录，必须使用另外一个XML元素：<codeBase>元素（该元素将会在本章后面详细介绍）。

使用分号定界符，privatePath特性可以指定多个子目录。现在并不需要这样做，但下面展示一个例子，它通知CLR去探测MyLibraries和MyLibraries\Tests这两个子目录：

```
<probing privatePath="MyLibraries;MyLibraries\Tests"/>
```

为了测试一下，请改变配置文件名称（随便取一个名字），然后再次运行程序。这一次程序将会运行失败。请记住*.config文件必须以相关的客户端应用程序名作为自己的前缀。最后再做一个测试，打开配置文件，更改其中任意XML元素的大小写。保存文件，你会发现你的程序再次不能正常运行（前面提到过，XML文件是区分大小写的）。

说明 要理解CLR在探查处理中会加载第一个找到的程序集。比如，如果C:\MyApp文件夹包含CarLibrary.dll的副本，它就会被加载到内存中，而MyLibraries中的副本则会被忽略。

14.5.4 App.Config文件

你当然可以使用文本编辑器手动编写XML配置文件，但Visual Studio 允许在开发客户程序的同时创建配置文件。首先，通过Project→Add New Item菜单选项向项目添加一个新的Application Configuration文件项。注意在图14-11中，该文件的名称为推荐的App.Config。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

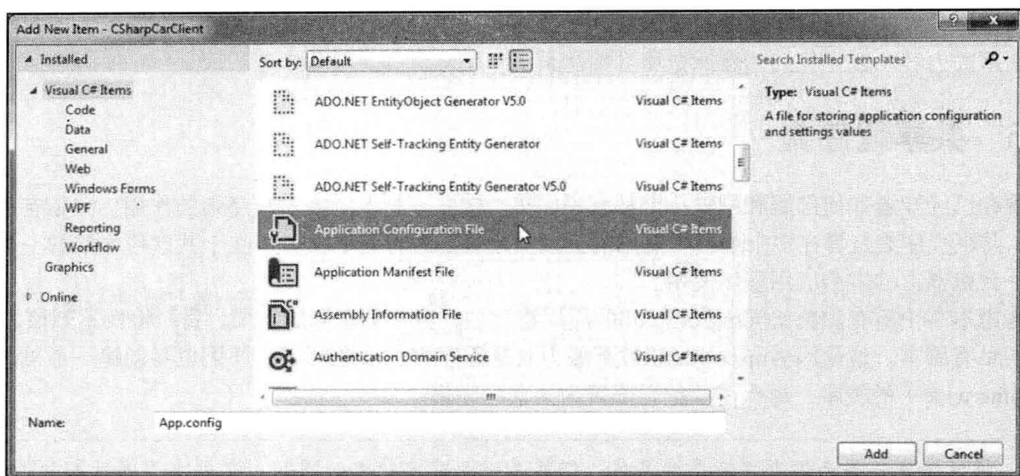


图14-11 在Visual Studio新项目中插入新的App.config文件

打开该文件，你会发现一组很少的指令，可以向其中添加其他元素：

现在，可以为创建的应用程序输入需要的XML元素。奇妙的事情发生了，每次编译项目时，Visual Studio会自动把App.config中的数据复制到\bin\Debug目录下的一个新文件中，并将它改为一个合适的名称（如CSharpCarClient.exe.config）。但是，请记住，做这一切的前提是该配置文件的名字必须是App.config，如图14-12所示。

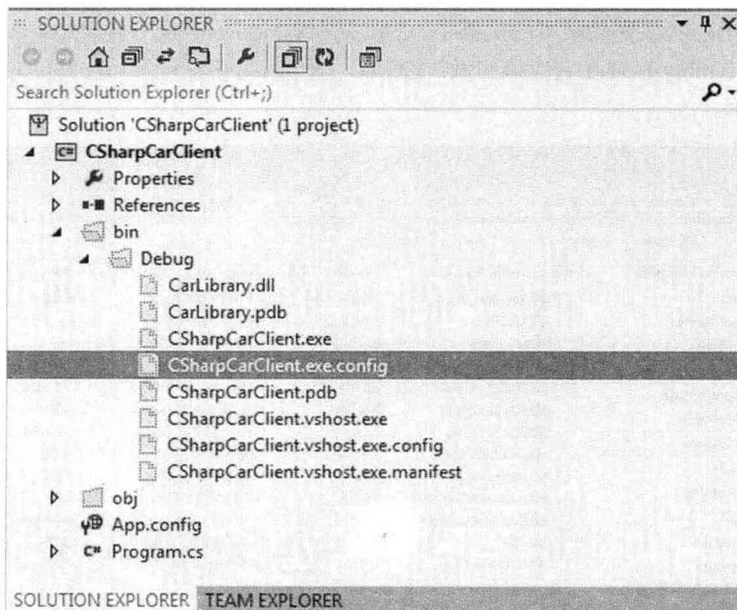


图14-12 App.config的内容将被复制到输出目录中正确命名的*.config中

使用这种方法，我们需要做的事情只是维护App.config的内容，Visual Studio会确保应用程序目录包含最新的内容（就算我们偶然地把项目重命名了）。

14.6　共享程序集

现在已经知道如何部署和配置一个私有程序集，接着开始了解共享程序集的作用。和私有程序集一样，共享程序集是旨在多个项目中复用的类型的集合。两者的本质区别在于共享程序集的一个副本可供一台机器上的多个应用程序使用。

考虑本书中所有需要访问mscorlib.dll的应用程序。查看一下它们的目录，我们将找不到该.NET程序集的私有副本。这是因为mscorlib.dll被部署为共享程序集。由此可见，如果想要创建一个机器级别（machine wide）的类库，那么需要把它部署为共享程序集。

说明　将代码库部署为私有库还是共享库，仍然是一个需要权衡的问题，它取决于项目本身的细节。一般来说，如果构建的库可被大量应用程序使用，共享程序集将十分有帮助，因为你可以轻松地部署一个新的版本（稍后将看到）。

14.6.1　全局程序集缓存

在前面提到，一个共享程序集并不部署在使用它的应用程序目录中。相反，共享程序集安装在GAC中。但是，GAC的确切位置则取决于目标机器上所安装的.NET平台的版本。

在没有安装.NET 4.0或更高版本的机器中，GAC是在Windows目录下名为Assembly的子目录中（例如，C:\Windows\assembly）。现在它应该叫做“老GAC”了，因为它只包含1.0、2.0、3.0和3.5版本的.NET库。如图14-13所示。

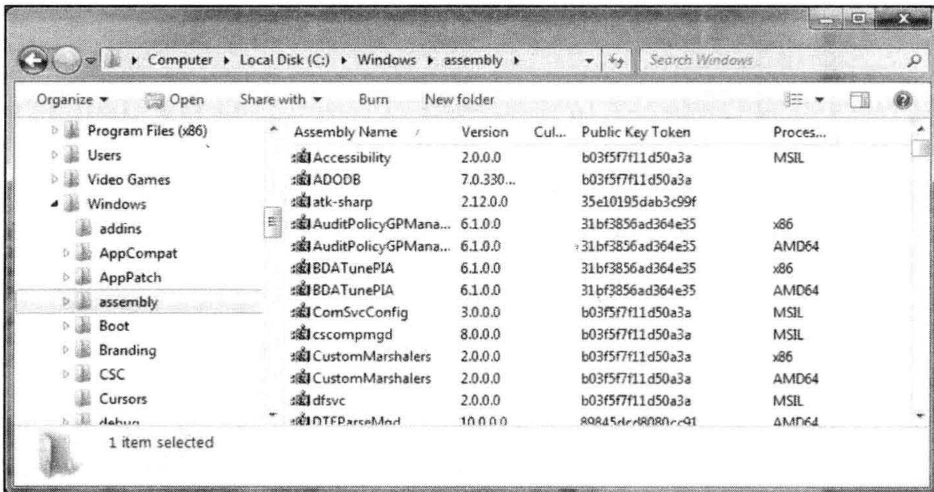


图14-13　老GAC

说明 不能把可执行的程序集（*.exe）安装到GAC。只有*.dll文件可以被部署为共享程序集。

.NET 4.0发布时，微软决定将.NET 4.0和更高版本的库隔离到单独的位置，即C:\Windows\Microsoft.NET\assembly\GAC_MSIL（如图14-14所示）。

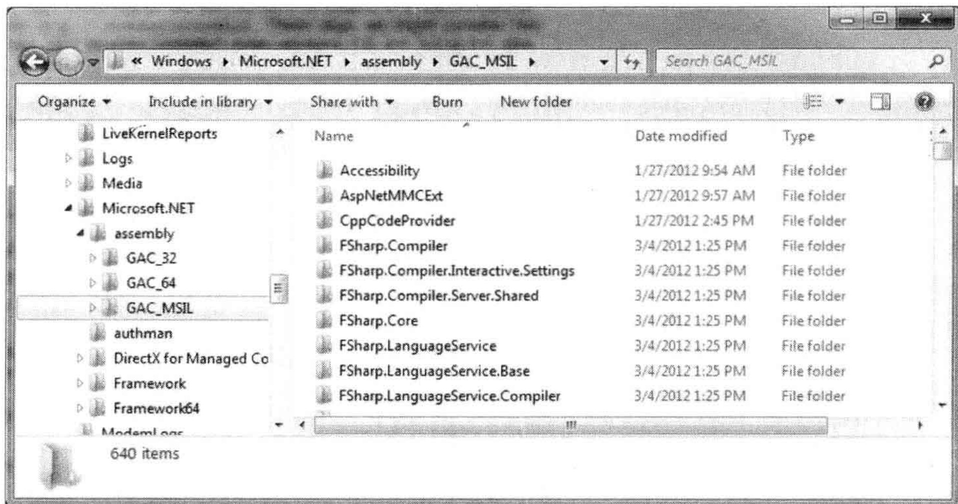


图14-14 .NET 4.0和更高版本的GAC

该新目录下存在很多子目录，都以特殊代码库的友好名称命名（如\System.WIndows.Forms和\System.Core等）。在给定的友好名称文件夹下，会发现另一个子目录，通常以下面的约定命名：

```
v4.0_major.minor.build.revision__publicKeyTokenValue
```

“v4.0”前缀表示该库由.NET 4.0或更高版本编译。该前缀后面是一个下划线，然后是该库的版本号（如1.0.0.0）。两个下划线之后，是另一串数字，叫做publickey标记值。在下一节将看到，这个公共键值是程序集“强名称”的一部分。最后，在这个文件夹下，你会发现当前讨论的*.dll文件的一个副本。

在本书中，我假设你都是使用.NET 4.5构建应用程序。因此如果要在GAC中安装库，将会安装到C:\Windows\Microsoft.NET\assembly\GAC_MSIL中。但要清楚的是，如果要配置使用3.5或更早版本编译的Class Library项目，共享库是安装在C:\Windows\assembly下的。

14.6.2 强名称

在部署程序集到GAC前，必须赋予它一个*强名称*。强名称用于标识给定.NET二进制文件的发行者。“发行者”可以是一个独立程序员（比如你），可以是某公司的一个部门，还可以是整个公司。

在一定程度上可以说，强名称在.NET中的作用好比全局唯一标识符（GUID）在COM中的作用。如果学过COM编程，你会想起AppID是标识特定COM应用程序的GUID。与COM GUID值（只是128位的二进制数字）不同，强名称是基于两个密码学上双关的钥（公钥和私钥）的，这种机制比起GUID更具唯一性和抗篡改性。

强名称由一组相关数据组成。我们可以使用以下程序集级别的特性来定义其中的大部分数据。

- 程序集的友好名称（程序集名称减去文件扩展名）。
- 程序集的版本号（使用[AssemblyVersion]特性赋值）。
- 公钥值（使用[AssemblyKeyFile]特性赋值）。
- 用于本地化的可选的区域性标识（使用[AssemblyCulture]特性赋值）。
- 嵌入的数字签名，使用基于程序集内容的散列值和私钥值生成。

为了给一个程序集赋予强名称，首先需要使用.NET Framework 4.5 的sn.exe工具生成公钥/私钥对。sn.exe工具生成一个文件[通常以*.snk（Strong Name Key）作为文件扩展名]，该文件包含两个不同的但算术上相关的钥：公钥和私钥。一旦C#编译器确定*.snk文件的位置，它就会在编译过程中使用.publickey标记把公钥值记录到程序集清单里。

另外，C#编译器还会产生一个基于整个程序集内容（CIL代码、元数据等）的散列值。在第6章已经讲过，散列码是对于某一固定输入的独一无二的数值输出。因此，如果更改了.NET程序集的内容（就算只是改变一个字符），编译器将产生完全不同的散列码。散列码结合*.snk文件中的私钥组成数字签名，并把它嵌入到程序集的CLR首部数据中。产生程序集的强名称的整个过程如图14-15所示。

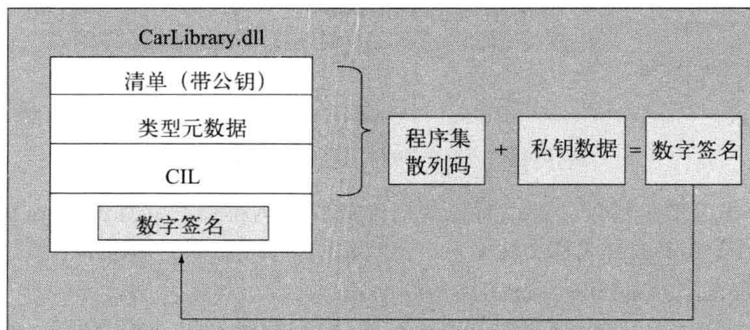


图14-15　编译期间，C#编译器将基于公钥和私钥数据生成数字签名，并把它嵌入到程序集中

实际的私钥并不在程序集清单中列出，它只用做对程序集内容进行数字签名（结合生成的散列码）。重申一点，使用公钥/私钥这个密码学概念，完全是为了确保在.NET领域中，没有任意两个公司、部门或个人拥有相同的标识。程序集被赋予强名称后，就可以安装到GAC中。

说明　强名称同时为程序集的内容提供了一定程度的保护，防止程序集被篡改。因此，.NET建议为每一个程序集（包括*.exe程序集）都赋予一个强名称，无论该程序集是否被安装到GAC。

14.6.3　在命令行生成强名称

让我们动手为先前创建的CarLibrary.dll程序集赋予强名称。现在，你可以使用Visual Studio生成需

要的*.snk文件，而在“万恶的旧社会”（大约2003年），为程序集进行强命名的唯一选择是命令行。我们来看看如何实现。

首先使用sn.exe工具生成需要的密钥。尽管sn.exe工具拥有众多命令行选项，但现在只需要关心-k标记，它会要求sn.exe工具生成一个包含公钥/私钥对的新文件。在硬盘C区创建一个名为MyTestKeyPair的文件夹，通过Developer Command Prompt转到该目录下。然后输入以下命令生成MyTestKeyPair.snk文件：

```
sn -k MyTestKeyPair.snk
```

公钥/私钥对生成后，接着需要把MyTestKeyPair.snk文件的位置告诉C#编译器。本章前面介绍过，如果使用Visual Studio创建任何新的C#项目，都会发现初始项目文件（位于Solution Explorer的Properties节点）中包含一个AssemblyInfo.cs文件。这个文件包含了大量描述程序集自身的特性。[AssemblyKeyFile]就是其中一个程序集级别的特性，它被添加到AssemblyInfo.cs文件中，以便告诉编译器有效的*.snk文件的位置所在。下面使用一个字符串来定义该位置：

```
[assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
```

说明 当手动指定[AssemblyKeyFile]特性时，Visual Studio会生成警告来通知你使用csc.exe的/keyfile选项，或者通过Visual Studio的Properties窗口建立密钥文件。不久你会使用IDE来完成（所以可以忽略警告）。

由于共享程序集的版本号也作为强名称的一部分，因此必须为CarLibrary.dll定义一个版本号。在AssemblyInfo.cs文件里，会看到另外一个名为[AssemblyVersion]的特性。先把它的值设为1.0.0.0：

```
[assembly: AssemblyVersion("1.0.0.0")]
```

.NET版本号由4个部分组成（*<major>.<minor>.<build>.<revision>*）。如果保持上面的设置，Visual Studio会自动在编译时增加版本值（使用*通配符标记，而非特定的版本值）。本例不需要这样做。思考如下代码：

```
// 格式：<Major number>.<Minor number>.<Build number>.<Revision number>
// 每一部分的有效取值范围都是0~65535
[assembly: AssemblyVersion("1.0.*")]
```

现在，C#编译器已经获取了所有用于生成强名称数据的信息（并没有通过[AssemblyCulture]特性设置独特的Culture属性值，而是自动“继承”了当前机器中的Culture属性值，在本例中为US English）。

编译CarLibrary代码库，然后使用ildasm.exe查看它的清单。会看到.publickey标签记录了公钥的所有信息，而.ver标记则记录了前面通过[AssemblyVersion]特性设置的版本号（如图14-16所示）。

这时，我们可以向GAC中部署共享的CarLibrary.dll程序集。不过要记住，现在.NET开发者可以使用Visual Studio中友好的用户界面创建强命名的程序集，而不用再使用神秘的sn.exe命令行工具。在查看详情之前，确保从AssemblyInfo.cs文件中删除（或注释掉）下面这行代码：

```
// [assembly: AssemblyKeyFile(@"C:\MyTestKeyPair\MyTestKeyPair.snk")]
```

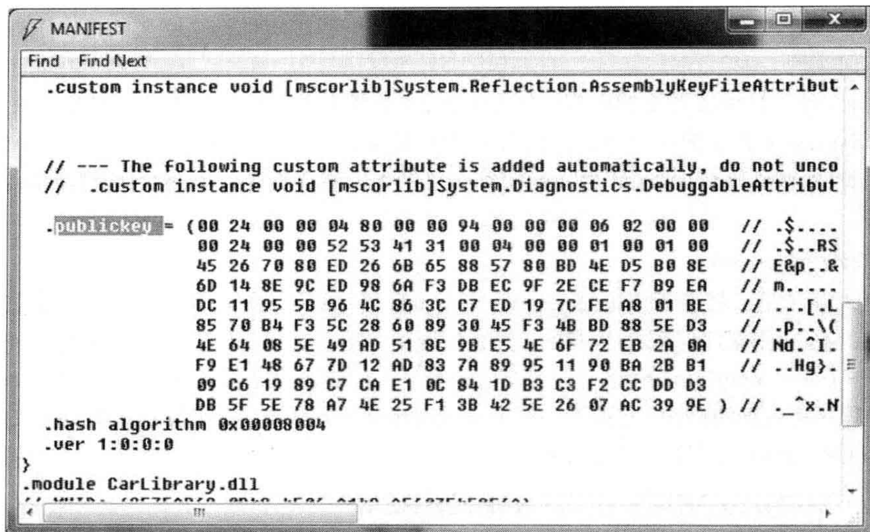


图14-16　强名称程序集在清单中记录了公钥

14.6.4　使用Visual Studio为程序集赋予强名称

Visual Studio允许我们使用项目的Properties页指定*.snk文件的位置，并生成新的*.snk文件。在CarLibrary项目中首先双击Solution Explorer中的Properties图标，选择Signing选项卡，然后选中Sign the assembly复选框并选择下拉列表中的<New...>选项（如图14-17所示）。

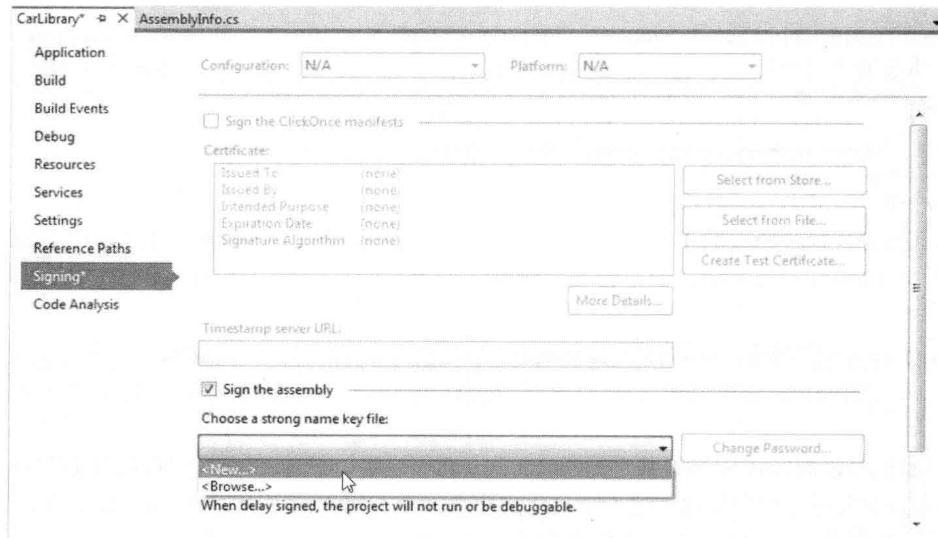


图14-17　使用Visual Studio创建新的*.snk文件

然后，你需要为新的*.snk文件提供一个名称（如myKeyFile.snk），而且你还可以选择是否用密码来保护文件（本例不需要这样做），如图14-18所示。



图14-18 使用Visual Studio命名*.snk文件

这时，你将在Solution Explorer中看到*.snk文件（如图14-19所示）。每次生成应用程序时，该文件中的数据都将为程序集分配一个合适的强名称。

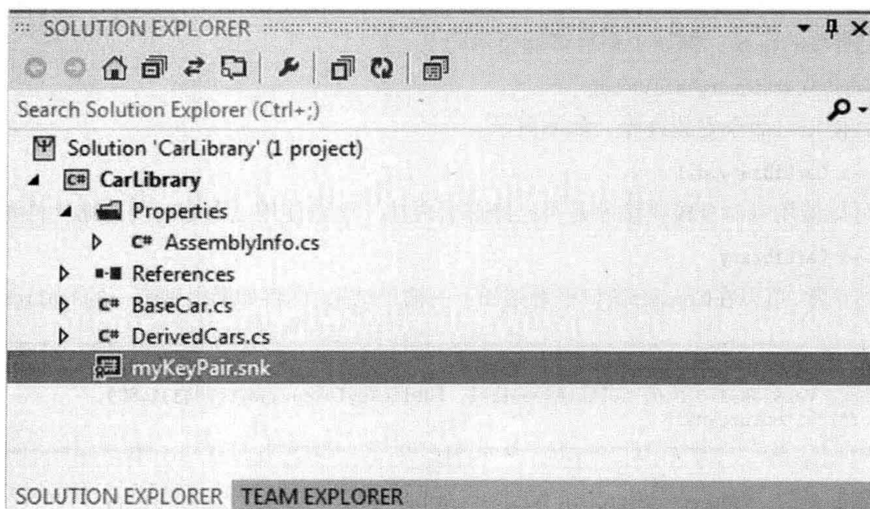


图14-19 Visual Studio在每次编译时都将为程序集进行强签名

说明 属性编辑器的Application标签页提供了一个名为Assembly Information（程序集信息）的按钮。单击这个按钮，会出现一个对话框，我们可以用来创建包含版本号、版权信息在内的许多程序集级别的特性。

14.6.5 在GAC中安装强名称的程序集

最后一步就是把CarLibrary.dll（现在它已经具有强名称）安装到GAC中。把程序集安装到GAC的最简单方法就是创建Windows MSI安装包（或者使用商用安装程序，比如InstallShield），.NET Framework 4.5 SDK提供了命令行工具gacutil.exe，它用于快速测试。

说明 你必须拥有管理员权限才能与机器中的GAC进行交互，这可能需要调整User Access Control（UAC，用户访问控制）的设置。

表14-1记录了gacutil.exe相关的参数选项（也可以在命令中使用/?标志来查看每个选项）。

表14-1 gacutil.exe的各种参数选项

选　项	作　用
-i	将强名称程序集安装入GAC
-u	从GAC中卸载程序集
-l	在GAC中显示程序集（或特定程序集）

要使用gacutil.exe安装强命名程序集，首先要打开Developer Command Prompt，然后将目录定位到CarLibrary.dll所在的目录，例如（路径可能会不同）：

```
cd C:\MyCode\CarLibrary\bin\Debug
```

接下来，使用-i命令安装该库，如下所示：

```
gacutil -i CarLibrary.dll
```

然后，可以使用-l命令核实该库是否已经部署成功（注意使用-l命令时要忽略文件扩展名）：

```
gacutil -l CarLibrary
```

如果一切正常，可以在Console窗口中看到如下的输出结果（你将如期得到唯一的PublicKeyToken值）：

```
The Global Assembly Cache contains the following assemblies:
Carlibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9,
processorArchitecture=MSIL
```

此外，导航到C:\Windwos\Microsoft.NET\assembly\GAC_MSIL，会发现包含正确子目录结构的新的CarLibrary文件夹（如图14-20所示）。

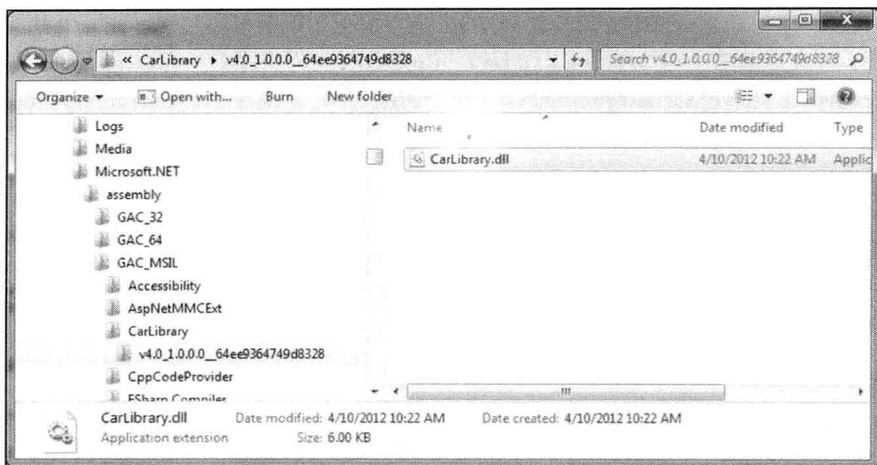


图14-20　在GAC中共享CarLibrary程序集

14.7 使用共享程序集

和使用私有程序集构建程序相比，使用共享程序集构建程序的唯一不同在于如何在Visual Studio中添加程序集引用。事实上，工具上没有任何改变（仍然使用Add Reference对话框）

如果需要引用私有程序集，可以使用Browse按钮导航到GAC中正确的子目录。不过，你还可以简单地导航到强命名程序集所在的位置（如某类库项目的/bin/debug文件夹），然后引用该副本。当Visual Studio找到一个强命名程序集时，将不会把该库拷贝到客户端应用程序的输出文件夹。图14-21展示了引用的库。

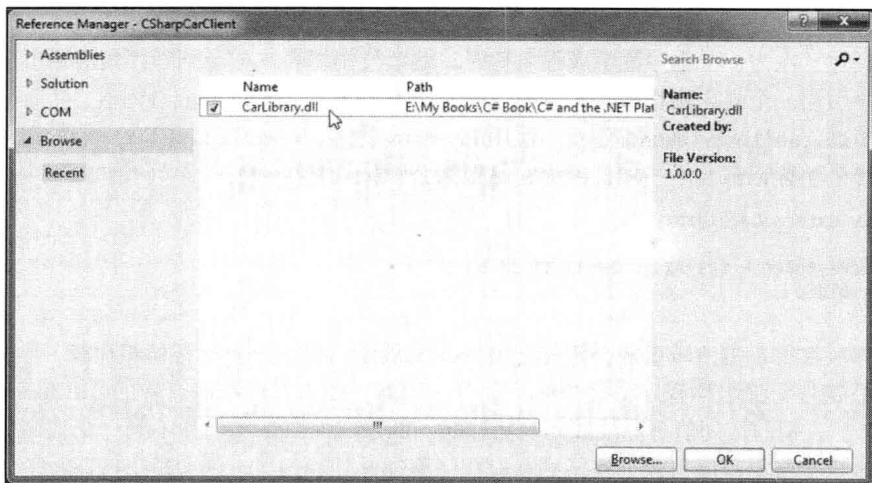


图14-21　使用Visual Studio引用强命名的共享CarLibrary（1.0.0.0版）

为了演示这一点，新建一个C# 控制台应用程序，命名为SharedCarLibClient，并引用前面描述的CarLibrary.dll程序集。现在，在Solution Explorer的Reference文件夹里有一个图标。选中该图标，然后查看Properties窗口（可通过Visual Studio的View菜单访问），会发现选中的CarLibrary的Copy Local属性设置为false。不管怎样，在新的客户端应用程序中编写如下的测试代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using CarLibrary;

namespace SharedCarLibClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Shared Assembly Client *****");
      SportsCar c = new SportsCar();
      c.TurboBoost();
      Console.ReadLine();
    }
  }
}
```

编译完客户应用程序后，使用Windows资源管理器查看含有SharedCarLibClient.exe文件的文件夹，会发现Visual Studio 并没有把CarLibrary.dll复制到客户应用程序的目录下。当发现引用程序集的清单中含有.publickey值时，Visual Studio就会假定这个具有强名称的程序集已被部署到GAC中，因此它将不会对该程序集进行复制工作。

查看SharedCarLibClient程序集的清单

前面讲到过，在为一个程序集生成强名称时，整个公钥将会被记录到程序集清单里面。而一个客户端应用程序引用一个强名称程序集时，.NET会把该程序集的公钥压缩成散列值，然后存放在客户端应用程序清单的.publickeytoken标签里。使用ildasm.exe查看SharedCarLibClient.exe的清单时，将会看到以下内容（公钥标记值会不一样，因为它是根据公钥值计算得到的）：

```
.assembly extern CarLibrary
{
  .publickeytoken = (33 A2 BC 29 43 31 E8 B9 )
  .ver 1:0:0:0
}
```

如果把存放在客户端清单里的公钥标记值与GAC展示给你的公钥标记值相比较，会发现它们完全匹配。记得公钥作为强名称程序集标识的一部分。因此，CLR只会加载这样的CarLibrary程序集：版本号为1.0.0.0，公钥的散列值是33A2BC294331E8B9。如果CLR在GAC中找不到一个程序集符合以上要求（同时在客户端应用程序目录下找不到私有程序集CarLibrary），则会抛出FileNotFoundException异常。

源代码 SharedCarLibClient应用程序的源代码位于Chapter 14子目录下。

14.8 配置共享程序集

和私有程序集一样，共享程序集也可以使用*.config文件进行配置。由于共享程序集安装在一个公共的地方（GAC），因此并不需要像私有程序集那样在配置文件中定义<privatePath>元素（如果客户程序既使用共享程序集又使用私有程序集，那么在*.config文件里<privatePath>元素还是有的）。

如果想要指示CLR绑定到某一程序集的其他版本，可以使用应用程序配置文件和共享程序集，我们就可以有效地跳过客户程序清单中记录的值。这样做的好处很多。比如，假设你已经发布了程序集（代码库）的1.0.0.0版本，但是后来发现一个重大的缺陷。一种补救措施是：重新生成客户端应用程序，使其引用已经消除缺陷的程序集版本（如1.1.0.0版本），然后把新的客户端应用程序和新的代码库发布到每一台目标机器上。

另一种做法是发布新的代码库，使用一个*.config文件在运行时自动指示运行库绑定到新的无错版。只要新版本的程序集已经安装到GAC中，原来的客户应用程序就不需要重新编译生成和发布。

再举一个例子：在发布了第一个消除缺陷的程序集（1.1.0.0版本）的一到两个月后，一些新的功能被添加到程序集中，由此而产生2.0.0.0版本。显然，目前的客户端应用程序是根据代码库的1.0.0.0版本进行编译的，因此它并不知道新版本程序集里面的新类型（假定在客户端应用程序代码里并没有引用到）。

此时，如果客户端应用程序想要使用2.0.0.0版本里面提供的新功能，在.NET下，可以把2.0.0.0版本的程序集发布到目标机器，让2.0.0.0版本和1.0.0.0版本和平共处。所以在需要的时候，只要修改应用程序的配置文件，然后原有的客户程序就可以动态地定向到2.0.0.0版本了（以便调用新的功能），这一切并不需要重新编译和部署。

14.8.1 冻结当前的共享程序集

为了说明如何动态绑定到共享程序集的某一个版本，请打开Windows资源管理器，然后把CarLibrary.dll当前版本（1.0.0.0）放到独立的目录（CarLibrary Version 1.0.0.0）中，如图14-22所示。

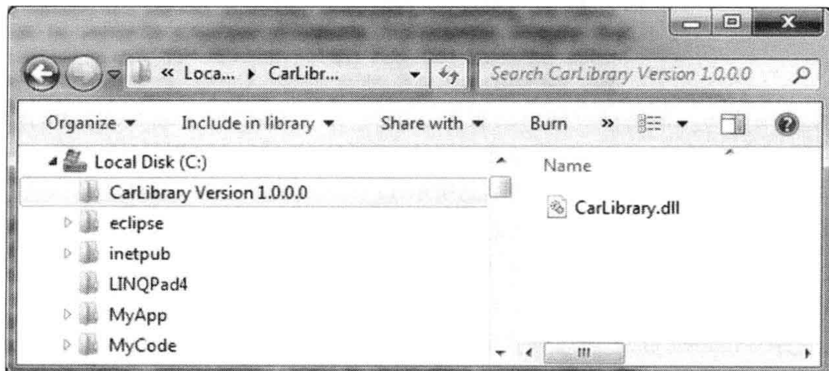


图14-22 冻结CarLibrary.dll的当前版本

14.8.2 构建共享程序集 2.0.0.0 版本

现在让我们来更新CarLibrary项目的内容。定义一个枚举类型MusicMedia，定义4种音乐播放器：

```
// 这辆拥有哪种音乐播放器
public enum MusicMedia
{
  musicCd,
  musicTape,
  musicRadio,
  musicMp3
}
```

接着，为Car类型添加一个新的公共方法，允许调用者打开某一个给定的音乐播放器（如果需要，一定要引入System.Windows.Forms命名空间）。如下所示：

```
public abstract class Car
{
...
  public void TurnOnRadio(bool musicOn, MusicMedia mm)
  {
    if(musicOn)
      MessageBox.Show(string.Format("Jamming {0}", mm));
    else
      MessageBox.Show("Quiet time...");
  }
}
```

更新Car类的构造函数，显示一个MessageBox，验证我们正在使用CarLibrary 2.0.0.0版本：

```
public abstract class Car
{
...
  public Car()
  {
    MessageBox.Show("CarLibrary Version 2.0!");
  }
  public Car(string name, int maxSp, int currSp)
  {
    MessageBox.Show("CarLibrary Version 2.0!");
    PetName = name; MaxSpeed = maxSp; CurrentSpeed = currSp;
  }
...
}
```

最后，在重新编译前，请确保程序集的版本号是2.0.0.0。双击Solution Explorer的Properties图标，单击Application选项卡中的Assembly Information...按钮。这样，就可以简单地更新Assembly Version的数值，以可视的方式来修改版本号（如图14-23所示）。

现在查看项目的\bin\Debug目录，会发现新版本的程序集（2.0.0.0），而1.0.0.0版本则还在CarLibrary Version 1.0.0.0子目录下。使用本章前面介绍的gacutil.exe把新版本安装到4.0 GAC中。注意，你现在有同一程序集的两个版本（如图14-24所示）。

如果现在使用Windows资源管理器执行SharedCarLibClient.exe，还不能看到显示“CarLibrary Version 2.0”的消息框，因为现在客户程序还是引用版本1.0.0.0程序集。我们应该如何指示CLR去绑定版本2.0.0.0呢？

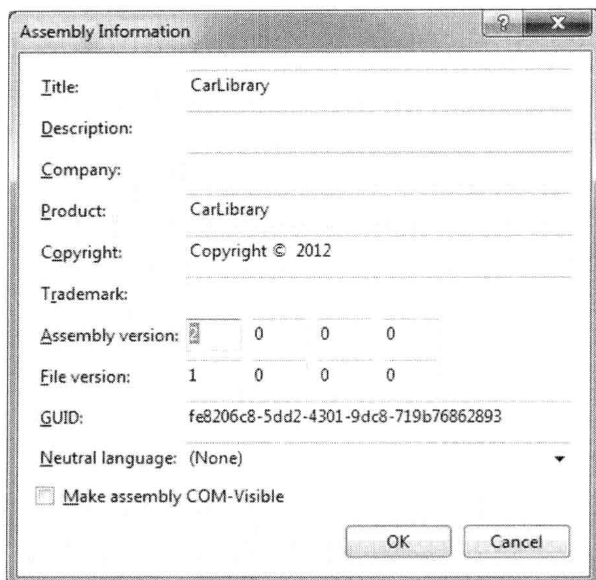


图14-23 设置CarLibrary.dll的版本号为2.0.0.0

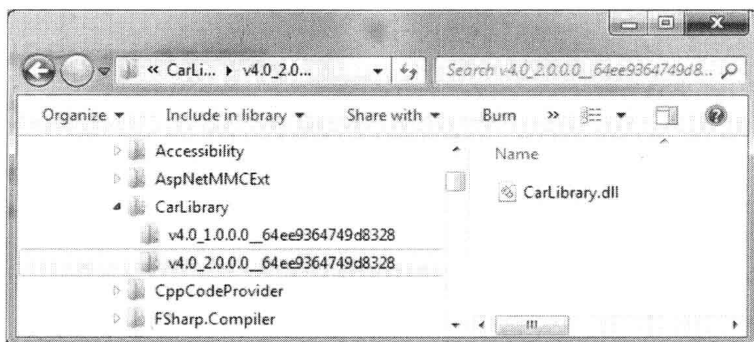


图14-24 共存的两个版本

说明 Visual Studio会在我们编译应用程序的时候自动重置引用！因此，当在Visual Studio中运行SharedCarLibClient.exe应用程序时，它会选择2.0.0.0版本的CarLibrary.dll！如果我们不小心这样运行程序的话，只要删除当前的CarLibrary.dll引用并选择1.0.0.0版本（我建议过你把它放在叫CarLibrary Version 1.0.0.0的文件夹中）就可以了。

14.8.3 动态重定向到共享程序集的特定版本

如果想要CLR加载一个不同于程序清单中的版本的共享程序集版本，需要使用*.config配置文件中

的<dependentAssembly>元素。在<dependentAssembly>元素里，需要创建<assemblyIdentity>子元素，它用于指定列在客户清单中的程序集的友好名称（如CarLibrary）和一个可选的区域性（culture）特性（如果想使用机器的默认区域性设置，则把该特性设置为空或者完全省略就可以了）。另外，<dependentAssembly>元素还需要创建<bindingRedirect>子元素，它用来定义程序清单当前指向的版本（使用子元素的oldVersion特性表示）和GAC中的替代版本（使用子元素的newVersion特性表示）。

修改SharedCarLibClient应用程序目录下名为SharedCarLibClient.exe.config的配置文件（文件具体内容如下，包含下面的XML数据）。

说明　公钥标记的值可能不同于下面的标记，要查看你的公共键标记值，可以在ildasm.exe中打开客户端，双击MANIFEST图标，将值拷贝到剪贴板（要移除空格）。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!--Runtime binding info -->
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture="neutral"/>
        <bindingRedirect oldVersion= "1.0.0.0"
                         newVersion= "2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

现在通过双击Windows资源管理器中的可执行文件来运行SharedCarLibClient.exe。程序将会弹出消息框以显示2.0.0.0被加载的信息。

程序配置文件中可以定义多个<dependentAssembly>元素。假定SharedCarLibClient.exe原来引用了MathLibrary程序集的2.5.0.0版本，而现在需要重定向到MathLibrary的3.0.0.0版本（同时CarLibrary重定向到2.0.0.0版本），则SharedCarLibClient.exe.config文件内容如下：

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
        <!- 绑定到CarLibrary的控件 -->
      <dependentAssembly>
        <assemblyIdentity name="CarLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture=""/>
        <bindingRedirect oldVersion= "1.0.0.0" newVersion= "2.0.0.0"/>
      </dependentAssembly>

        <!- 绑定到MathLibrary的控件 -->
      <dependentAssembly>
        <assemblyIdentity name="MathLibrary"
                          publicKeyToken="64ee9364749d8328"
                          culture=""/>
        <bindingRedirect oldVersion= "2.5.0.0" newVersion= "3.0.0.0"/>
      </dependentAssembly>
```

```
    </assemblyBinding>
  </runtime>
</configuration>
```

说明 可以通过oldVersion特性指定一系列老版本号。例如，<bindingRedirect oldVersion="1.0.0.0-1.2.0.0" newVersion="2.0.0.0"/>告诉CLR对于任何在1.0.0.0到1.2.0.0之间的老版本都使用2.0.0.0版本。

14.9 发行者策略程序集

下一个关于配置的话题是发行者策略程序集（Publisher Policy Assembly）的作用。我们已经知道，使用*.config文件可以忽略清单中记录的版本，并使客户端程序绑定到指定版本的共享程序集。这一切看上去都非常美好，但请想象一下，一个管理员需要做的工作是：重新配置所有的客户端程序，使其绑定到CarLibrary.dll程序集的2.0.0.0版本。由于配置文件的命名要求非常严格，因此需要把相同的XML内容复制到所有的客户端（而能够这样做，还需要知道所有使用CarLibrary程序集的客户端目录位置）。很明显，这是一场噩梦。

发行者策略允许程序集的发行者（用户、用户的部门、用户的公司等）在安装相关的最新版本程序集到GAC的同时，把一个*.config文件的二进制版本也安装到GAC。这样做的好处是客户端应用程序目录不需要包含任何*.config文件。CLR会读取当前客户端程序的清单，尝试在GAC中查找被请求的版本。但如果CLR找到一个发行者策略程序集，它会读取其中嵌入的XML数据，在GAC级别执行请求重定向。

我们使用.NET的al.exe工具在命令行创建发行者策略程序集。尽管该工具拥有众多选项，但创建一个发行者策略程序集只需要使用到以下几个参数。

- 含有重定向指令的*.config或者*.xml文件的位置。
- 生成的发行者策略程序集的名称。
- 用于对发行者策略程序集签名的*.snk文件的位置。
- 创建的发行者策略程序集的版本号。

以下命令集用于创建控制CarLibrary.dll的发行者策略程序集（这必须在命令行窗口的单独一行内输入）：

```
al /link: CarLibraryPolicy.xml /out:policy.1.0.CarLibrary.dll
/keyf:C:\MyKey\myKey.snk /v:1.0.0.0
```

这里，XML内容被包含在CarLibraryPolicy.xml文件中。命令输出的文件名（格式必须为：*policy.<major>.<minor>.assemblyToConfigure*）由/out标志指定。另外包含公钥/私钥对的文件名由/keyf选项指定（记住，因为发行者策略文件是共享的，因此它必须具有强名称）。

命令执行完以后，一个新的程序集可以被安装到GAC中，它将强制要求所有客户端程序绑定到CarLibrary.dll的2.0.0.0版本，而不需要使用特定的客户端应用程序配置文件。由此可以通过已有程序集的给定版本（或版本范围）为所有应用程序设计“机器级别”的重定向。

禁止发行者策略

现在，假定你（作为系统管理员）需要部署一个发行者策略程序集（和最新版本的相关程序集）到一台客户端机器上。如果比较幸运的话，90%的受影响的客户端应用程序会正确地重绑定到2.0.0.0版本。先不管原因，假定剩下的10%的客户端应用程序在使用CarLibrary.dll 2.0.0.0版本时发生了错误（我们知道，向后兼容的软件并不是总能100%地正常工作）。

在这种情况下，针对每个有问题的客户端应用程序创建一个配置文件，通过该文件通知CLR忽略安装在GAC上的发行者策略程序集。那些其余的能够正常工作的客户端程序则接受发行者策略程序集指定的重定向，它们使用最新版本的程序集。为了逐个客户端程序地禁止发行者策略，我们需要创建一个使用<publisherPolicy>元素的*.config文件，同时把该元素的apply特性设置为no。现在，CLR将会加载客户端程序清单中原来指定的程序集版本。

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <publisherPolicy apply="no" />
    </assemblyBinding>
  </runtime>
</configuration>
```

14.10　<codeBase>元素

应用程序配置文件也可以指定代码库。<codeBase>元素用于指示CLR探测位于任意位置（例如网络终点，或者客户端应用程序目录以外的本地目录）的依赖程序集。

当<codeBase>元素的值指向远程计算机的时候，相关程序集将会按需下载到GAC的下载缓存中。由于GAC对部署其中程序集的要求，所以那些通过<codeBase>元素加载的程序集应该具有强名称（否则，CLR无法把远程程序集加载到GAC）。我们可以使用gacutil.exe结合/idl选项查看下载缓存中的内容。如下所示：

```
gacutil /ldl
```

说明　实际上，<codeBase>元素可以用于探测不具有强名称的程序集。但是，该程序集的位置必须是相对于客户端应用程序目录的（这一点与<privatePath>元素有点儿不同）。

为了研究运行时的<codeBase>，创建控制台应用程序CodeBaseClient，添加对CarLibrary.dll 2.0.0.0版本的引用，然后改写生成的代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using CarLibrary;
```

```
namespace CodeBaseClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Fun with CodeBases *****");
      SportsCar c = new SportsCar();
      Console.WriteLine("Sports car has been allocated.");
      Console.ReadLine();
    }
  }
}
```

由于CarLibrary.dll被部署在GAC中，所以客户端程序可以正常运行。然而，为了说明<codeBase>元素，我们在C盘下创建一个新的文件夹（C:\MyAsms），然后把CarLibrary.dll的2.0.0.0版本复制到该目录下。

向CodeBaseClient项目前面已介绍过添加App.config文件（或编辑已有的App.Config），然后编写以下XML内容（记住.publickeytoken的值会和读者的有所不同，请到GAC中查看）：

```
<configuration>
...
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name=" CarLibrary" publicKeyToken="33A2BC294331E8B9" />
        <codeBase version="2.0.0.0" href="file:///C:/MyAsms/CarLibrary.dll" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

可以看到，<codeBase>元素内嵌在<assemblyIdentity>元素中。<assemblyIdentity>元素使用name特性和publicKeyToken特性来指定友好名称和对应的publicKeyToken的值。<codeBase>元素则指定需要加载的程序集的版本和位置（通过href属性）。现在，如果只把GAC中的CarLibrary.dll的2.0.0.0版本删除，则客户端程序仍然可以正常运行，因为CLR在C:\MyAsms下能够找到被请求的程序集。

说明 如果在开发机器上随意放置程序集，则等同于为自己重新创建了一个系统注册表（这将会带来非常闻名的"DLL地狱"）。因为一旦移动或重命名程序集所在目录，相关应用程序将运行失败。因此，请谨慎使用<codeBase>。

<codeBase>元素能够引用位于远程网络计算机上的程序集。假设你拥有足够的权限访问http://www.MySite.com。为了把远程的*.dll下载到本地机器上的GAC下载缓存区，可以对<codeBase>元素做如下修改：

```
<codeBase version="2.0.0.0"
  href="http://www.MySite.com/Assemblies/CarLibrary.dll" />
```

源代码 CodeBaseClient应用程序的源代码位于Chapter 14子目录下。

14

14.11　System.Configuration 命名空间

到目前为止，本章中所有的*.config文件都使用了多个常用的XML元素。CLR能够读取这些元素来解析外部程序集的位置。除了这些元素之外，在客户端配置文件中，我们还能够增加基于具体应用程序的自定义信息。为此，.NET Framework提供了一个命名空间，这使得我们可以以编程方式读取配置文件的这些数据。

System.Configuration命名空间提供了一组类型供开发人员读取*.config文件的自定义数据。这些自定义的配置信息必须放在<appSettings>元素里。<appSettings>元素可包含任意多个<add>元素（用于定义键/值对）供开发人员以编程方式获取。

例如，假定控制台应用程序AppConfigReaderApp拥有一个App.config文件，其中定义了两个应用程序特定的值：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>

  <!-- 自定义应用设置 -->
  <appSettings>
    <add key="TextColor" value="Green" />
    <add key="RepeatCount" value="8" />
  </appSettings>
</configuration>
```

客户端应用程序只需要调用System.Configuration.AppSettingsReader类的实例方法GetValue()，就可以读取这些信息。在下面的代码中可以看到，GetValue()方法的第一个参数是*.config文件中键的名称，第二个参数则是该键的类型（使用C#的typeof操作符获取）：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Text;

using System.Configuration;

namespace AppConfigReaderApp
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Reading <appSettings> Data *****\n");

    // 从*.config文件获取自定义数据
      AppSettingsReader ar = new AppSettingsReader();
      int numbOfTimes = (int)ar.GetValue("RepeatCount", typeof(int));
      string textColor = (string)ar.GetValue("TextColor", typeof(string));

      Console.ForegroundColor =
        (ConsoleColor)Enum.Parse(typeof(ConsoleColor), textColor);
```

```
    // 现在正确输出消息
      for (int i = 0; i < numbOfTimes; i++)
        Console.WriteLine("Howdy!");
      Console.ReadLine();
    }
  }
}
```

源代码 AppConfigReaderApp应用程序的源代码位于Chapter 14子目录下。

14.12 配置文件架构文档

本章介绍了XML配置文件的作用。在这里，我们来看看可以向`<runtime>`元素添加一些设置，控制CLR如何定位到外部所需的库。在阅读本书后面章节（以及阅读完毕开始构建大规模软件）的时候，你很快会发现使用XML配置文件是家常便饭。

没错，.NET平台在很多API中都使用了*.config文件。例如在第25章，你将看到WCF（Windows Communication Foundation）使用了配置文件来建立复杂的网络设置。在本书后面介绍ASP.NET Web开发时，你很快会注意到web.config文件与桌面App.config文件含有相同类型的指令。

由于给定的.NET配置文件可以包含很多指令，你应该意识到在.NET帮助系统里，可以找到整个XML文件的架构。如果在帮助系统中搜索“Configuration File Schema for the .NET Framework”，将找到每个元素的详细解释（如图14-25所示）。

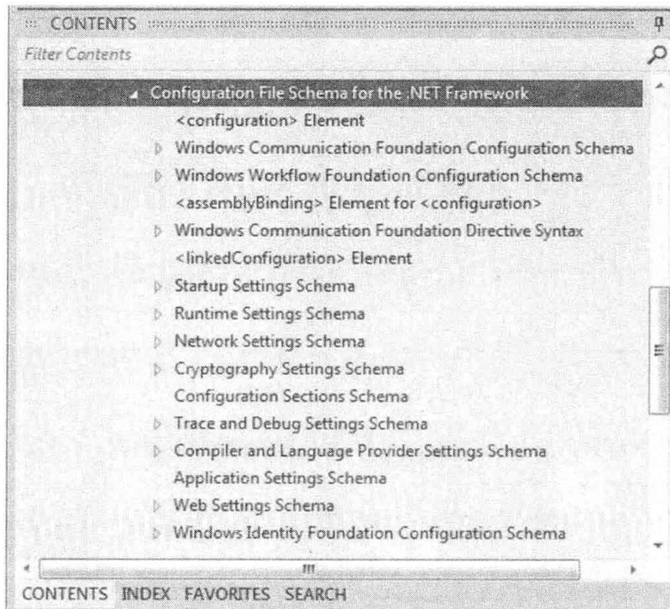


图14-25 XML配置文件完全位于.NET帮助系统中

14.13　小结

本章介绍了.NET类库（即.NET *.dll）的作用。如你所见，类库是包含可在多个项目中复用的逻辑的.NET二进制文件。库的部署有两种主要的方式：私有的或共享的。私有程序集部署到客户端文件夹或子目录，只要有适当的XML配置文件。共享程序集是机器中的所有应用程序都能使用的库，它同样也受客户端配置文件设置的影响。

我们学习了如何将共享程序集标记为“强名称”，从CLR的角度看，它实际上是建立了一个唯一标识的库。同样，我们还学习了不同的命令行工具（如sn.exe和gacutil.exe），用于共享库的开发和部署。

本章最后介绍了发布者策略以及实用System.Configuration命名空间存储和获取自定义配置的过程。

第 15 章

类型反射、晚期绑定和基于特性的编程

如第14章讲到的，在.NET中，程序集是基本的部署单元。使用集成在Visual Studio中的对象浏览器（和其他IDE），可以查看项目所引用程序集的类型。此外，对于一个.NET二进制文件，使用外部工具（如ildasm.exe）可以查看底层的CIL代码、类型元数据和程序集清单。除了在设计时对.NET程序集进行研究外，也可以使用System.Reflection命名空间通过编程获取相同的信息。本章的第一个任务就是明确反射的作用以及理解.NET元数据的必要性。

本章的剩余部分讨论了许多与反射服务密切相关的主题。举例来说，我们将学习.NET客户端如何使用动态加载和晚期绑定来激活在编译时未知的类型。我们也将学习如何使用系统提供的或自定义的特性来将自定义元数据插入到.NET程序集中。在讲述了所有这些（表面深奥的）主题之后，本章将以如何建立“插件对象”结束，读者可以将它们插入到一个可扩展的桌面GUI应用程序中。

15.1 类型元数据的必要性

使用元数据完整地描述类型（类、接口、结构、枚举和委托）的能力是.NET平台的一个关键要素。许多.NET技术，如WCF和对象序列化都需要这个能力在运行时发现类型格式。另外，跨语言互操作、编译器服务以及集成开发环境的智能感知能力都依赖于对类型的具体描述。

回想一下，ildasm.exe工具可以使用Ctrl+M组合键选项查看一个数据集的类型元数据（见第1章）。因此，如果使用ildasm.exe并按Ctrl+M组合键打开本书中建立的任意的*.dll或*.exe程序集（比如第14章创建的CarLibrary.dll），都将发现有关的类型元数据（如图15-1所示）。

可以看到，ildasm.exe中显示的.NET类型元数据非常详细（实际上二进制格式更紧凑）。事实上，如果列出CarLibrary.dll程序集的全部元数据描述，需要用好几页。这太浪费时间（也费纸），所以我们只看看CarLibrary.dll程序集中的主要元数据描述。

说明 不必太在意接下来的几节中出现的.NET元数据的确切语法。重点是.NET元数据非常具有描述性，它列出了给定代码库中每个定义的内部类型（和引用的外部类型）。

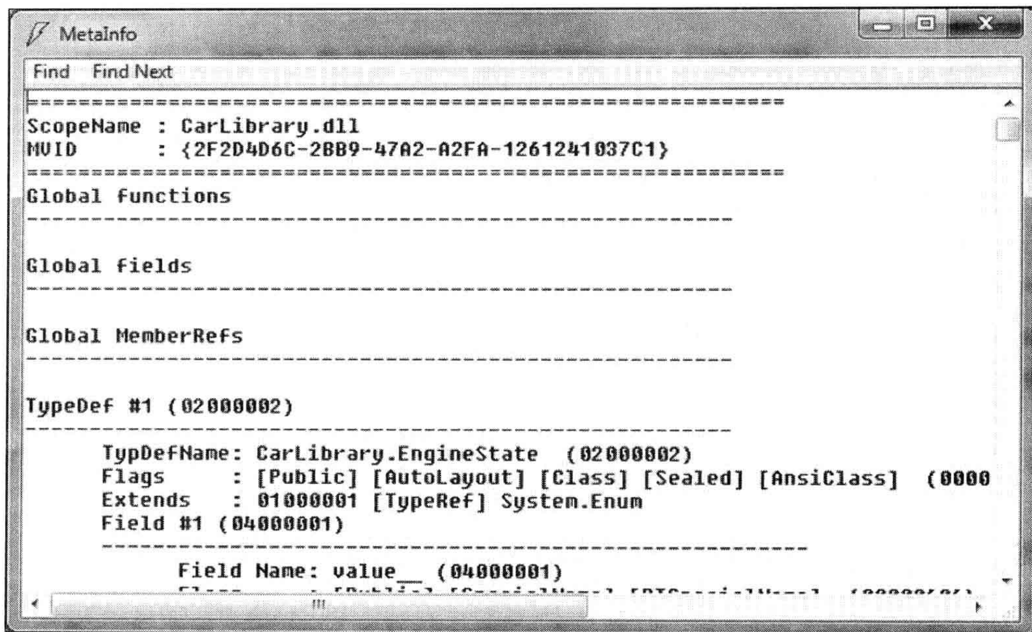


图15-1　使用ildasm.exe查看程序集元数据

15.1.1　查看（部分）EngineState 枚举的元数据

在当前程序集中定义的每个类型都使用一个TypeDef #n标记（这里，TypeDef是type definition的缩写）。如果使用一个独立的.NET程序集定义的类型来描述类型，被引用的类型将使用TypeRef #n标记。TypeRef（type reference的缩写）标记是一个指针，它指向外部程序集中被引用类型的全部元数据定义。简言之，.NET元数据是一组清晰地标记了所有的类型定义（TypeDefs）和被引用类型（TypeRefs）的表，所有这些都可以使用ildasm.exe元数据窗口查看到。

在CarLibrary.dll中，我们遇到的一个TypeDef是对CarLibrary.EngineState枚举的元数据描述（读者的编号可能不同，TypeDef编号是按照C#编译器处理文件的顺序进行的）：

```
TypeDef #2 (02000003)
-------------------------------------------------------
  TypDefName: CarLibrary.EngineState  (02000003)
  Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]  (00000101)
  Extends   : 01000001 [TypeRef] System.Enum
  Field #1 (04000006)
-------------------------------------------------------
    Field Name: value__ (04000006)
    Flags     : [Public] [SpecialName] [RTSpecialName]  (00000606)
    CallCnvntn: [FIELD]
    Field type:  I4

  Field #2 (04000007)
-------------------------------------------------------
    Field Name: engineAlive (04000007)
```

```
    Flags     : [Public] [Static] [Literal] [HasDefault]  (00008056)
    DefltValue: (I4) 0
    CallCnvntn: [FIELD]
    Field type:  ValueClass CarLibrary.EngineState
...
```

这里，TypDefName标记用来建立给定类型的名称，这里是自定义的CarLibrary.Engine枚举。而Extends元数据标记用来记录给定.NET类型的基类型（在本例中，就是被引用的类型System.Enum）。每个枚举中的字段使用Field #n来标记。为简洁起见，我只是列出了CarLibrary.EngineState.engineAlive的元数据。

15.1.2 查看（部分）Car类型的元数据

这里是Car类的部分转储信息，它阐释了以下几点。

- 如何用.NET元数据定义字段。
- 如何通过.NET元数据说明方法。
- 如何在.NET元数据中表示一个自动属性。

```
TypeDef #3 (02000004)
-------------------------------------------------------
  TypDefName: CarLibrary.Car  (02000004)
  Flags : [Public] [AutoLayout] [Class] [Abstract]
          [AnsiClass] [BeforeFieldInit]  (00100081)
  Extends : 01000002 [TypeRef] System.Object
...

  Field #2 (0400000a)
-------------------------------------------------------
  Field Name: <PetName>k__BackingField (0400000A)
  Flags     : [Private]  (00000001)
  CallCnvntn: [FIELD]
  Field type:  String

...

Method #1 (06000001)
-------------------------------------------------------
  MethodName: get_PetName (06000001)
  Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]  (00000886)
  RVA       : 0x000020d0
  ImplFlags : [IL] [Managed]  (00000000)
  CallCnvntn: [DEFAULT]
  hasThis
  ReturnType: String
  No arguments.

...

  Method #2 (06000002)
-------------------------------------------------------
  MethodName: set_PetName (06000002)
  Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]  (00000886)
  RVA       : 0x000020e7
  ImplFlags : [IL] [Managed]  (00000000)
  CallCnvntn: [DEFAULT]
```

```
    hasThis
    ReturnType: Void
    1 Arguments
      Argument #1:  String
    1 Parameters
      (1) ParamToken : (08000001) Name : value flags: [none] (00000000)
...

Property #1 (17000001)
-------------------------------------------------------
    Prop.Name : PetName (17000001)
    Flags     : [none] (00000000)
    CallCnvntn: [PROPERTY]
    hasThis
    ReturnType: String
    No arguments.
    DefltValue:
    Setter    : (06000002) set_PetName
    Getter    : (06000001) get_PetName
    0 Others
 ...
```

首先，注意Car类元数据标记了类型的基类（System.Object），并且包括了描述该类型如何被构造的多个标记（比如[Public]、[Abstract]等）。而方法（比如Car的构造函数）描述了它的参数、返回值和名称。

注意，在本例中，自动属性如何使编译器生成了一个私有的支持字段（名为<PetName>k__BackingField）和两个方法（如果是读/写属性的话）get_PetName()和set_PetName()。最后，注意如何使用.NET元数据的Setter/Getter标记将属性映射到它们内部的获取方法/设置方法上。

15.1.3　研究TypeRef

前面已经说到，程序集的元数据不仅可以描述一组内部类型（比如Car、EngineState等），而且也可以描述任何一个被内部类型引用的外部类型。举例来说，CarLibrary.dll定义了两个枚举，可以找到System.Enum类型的TypeRef块，如下所示：

```
TypeRef #1 (01000001)
-------------------------------------------------------
Token:              0x01000001
ResolutionScope:    0x23000001
TypeRefName:        System.Enum
```

15.1.4　记录定义的程序集

ildasm.exe元数据窗口中可以看到使用Assembly标记描述程序集自身的.NET元数据。如下面（部分）列表中所示，在Assembly表中记录的信息与通过MANIFEST图标看到的信息是相同的（太令人惊讶了）。这里是CarLibrary.dll清单中的部分转储信息（版本2.0.0.0）：

```
Assembly
-------------------------------------------------------
    Token: 0x20000001
    Name : CarLibrary
    Public Key    : 00 24 00 00 04 80 00 00 // 等等
```

```
Hash Algorithm : 0x00008004
Major Version: 0x00000002
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
Flags : [PublicKey] ...
```

15.1.5 记录引用的程序集

除了Assembly标记、TypeDef和TypeRef块组外，.NET元数据也使用AssemblyRef #n标记来记录每个外部的程序集。由于CarLibrary.dll使用了System.Windows.Forms.MessageBox类，所以可以找到System.Windows.Forms的AssemblyRef，例如：

```
AssemblyRef #2 (23000002)
-------------------------------------------------------
  Token: 0x23000002
  Public Key or Token: b7 7a 5c 56 19 34 e0 89
  Name: System.Windows.Forms
  Version: 4.0.0.0
  Major Version: 0x00000004
  Minor Version: 0x00000000
  Build Number: 0x00000000
  Revision Number: 0x00000000
  Locale: <null>
  HashValue Blob:
  Flags: [none] (00000000)
```

15.1.6 记录字符串字面量

关于.NET元数据，最后要提的一点是，事实上代码中的每个字符串字面量都记录在User Strings标记下，例如：

```
User Strings
-------------------------------------------------------
70000001 : (11) L"Jamming {0}"
70000019 : (13) L"Quiet time..."
70000035 : (23) L"CarLibrary Version 2.0!"
70000065 : (14) L"Ramming speed!"
70000083 : (19) L"Faster is better..."
700000ab : (16) L"Time to call AAA"
700000cd : (16) L"Your car is dead"
```

说明 如这段元数据代码所示，要始终注意所有的字符串都清晰地记录在程序集元数据中。如果使用字符串字面量来获取密码、信用卡号或其他敏感信息，将带来很大的安全问题。

15

下一个关注的问题可能是（最好的情形）“我如何能在我的应用程序中使用这些信息”，或者是（最差的情形）“为什么我需要关注元数据”。为了回答这些问题，我们需要介绍一下.NET反射服务。虽然接下来（直到本章最后）的内容或许有些令人头痛，但是它们真的非常有用，所以请跟紧些。

说明 你也会在MetaInfo窗口的显示中发现许多CustomAttribute标记，它记录了特性在代码库中的应用。在本章稍后会学习.NET特性的作用。

15.2 反射

在.NET中，反射（reflection）是一个运行库类型发现的过程。使用反射服务，可以通过编程使用一个友好的对象模型得到与通过ildasm.exe显示的相同的元数据信息。举例来说，通过反射，可以得到一个给定*.dll或*.exe程序集所包含的所有类型的列表，这个列表包括给定类型定义的方法、字段、属性和事件。也可以动态发现一组给定类型支持的接口、方法的参数和其他相关细节（基类、命名空间、清单数据等）。

与其他命名空间一样，System.Reflection（定义在mscorlib.dll）包含了大量相关类型。表15-1列出了常用的一些核心类型。

表15-1　System.Reflection命名空间成员示例

类　型	作　用
Assembly	该抽象类包含了许多静态方法，通过它可以加载、了解和操纵一个程序集
AssemblyName	使用该类可以找到大量隐藏在程序集的身份中的细节（版本信息、区域信息等）
EventInfo	该抽象类保存给定事件的信息
FieldInfo	该抽象类保存给定字段的信息
MemberInfo	该类是抽象基类，它为EventInfo、FieldInfo、MethodInfo和PropertyInfo类型定义了公共的行为
MethodInfo	该抽象类包含给定方法的信息
Module	该抽象类使你可以访问多文件程序集中的给定模块
ParameterInfo	该类保存给定参数的信息
PropertyInfo	该抽象类保存给定属性的信息

要理解如何使用System.Reflection命名空间编程读取.NET元数据，首先需要理解System.Type类。

15.2.1 System.Type类

System.Type类定义了很多成员，可以用来检查某个类型的元数据，它们返回的类型大多位于System.Reflection命名空间中。举例来说，Type.GetMethods()返回一个MethodInfo类型的数组，Type.GetFields()返回一个FieldInfo类型的数组等。System.Type提供的完整的成员组是很容易扩展的，表15-2提供了部分由System.Type支持的成员（详见.NET Framework 4.5 SDK文档）。

表15-2 System.Type的部分成员

类 型	作 用
IsAbstract、IsArray、IsClass、IsCOMObject、IsEnum、IsGenericTypeDefinition、IsGenericParameter、IsInterface、IsPrimitive、IsNestedPrivate、IsNestedPublic、IsSealed、IsValueType	这些属性（还有其他的）允许我们发现许多所引用类型的基本特性（比如，它是否是抽象方法、数组、嵌套类等）
GetConstructors()、GetEvents()、GetFields()、GetInterfaces()、GetMembers()、GetMethods()、GetNestedTypes()、GetProperties()	这些方法（还有其他的）允许我们得到表示感兴趣项目（接口、方法、属性等）的数组。每个方法返回一个相关数组（举例来说，GetFields()返回一个FieldInfo数组，GetMethods()返回一个MethodInfo数组等）。要知道每个方法都有单数的版本（比如，GetMethod()、GetProperty()等），可以通过名称得到指定的项，而不是得到有关项的数组
FindMembers()	该方法根据查询条件返回一个MemberInfo类型的数组
GetType()	该静态方法返回一个Type实例，给定一个字符串名称
InvokeMember()	该方法允许对给定项目的晚期绑定。本章后面会详细介绍“晚期绑定”

15.2.2 使用System.Object.GetType()得到Type引用

可以用多种方法得到一个Type类的实例。但是，由于Type是一个抽象类，所以不能直接使用new关键字创建一个Type对象。对此我们的首选是：使用System.Object定义的GetType()方法，它返回了一个表示当前对象元数据的Type类的实例：

```
// 使用一个SportsCar实例得到类型信息
SportsCar sc = new SportsCar();
Type t = sc.GetType();
```

显而易见，要想使用这个方法，必须得到类型的编译时信息（这里是SportsCar类），并且当前在内存中有类型实例。由于无法对自定义的程序集进行编译，类似ildasm.exe的工具并没有通过直接调用System.Object.GetType()来得到每个类型的类型信息。

15.2.3 使用typeof()得到Type引用

另一个获取类型信息的方法是使用C# typeof操作符，如下所示：

```
// 使用typeof得到类型
Type t = typeof(SportsCar);
```

类似System.Object.GetType()，使用typeof操作符，我们不需要先建立一个实例来提取类型信息。但是，仍然需要知道类型的编译时信息，因为typeof需要的是类型的强类型名称，而不是文本表示。

15.2.4 使用System.Type.GetType()得到Type引用

为了以更灵活的方式得到类型信息，我们可以调用System.Type类的静态成员GetType()，然后指定类型的完全限定名。采用这种方法，我们不需要得到正从中提取元数据的类型的编译时信息，假如

Type.GetType()取无所不在的System.String的实例。

说明 我说的“对于调用Type.GetType()，我们不需要编译时的知识”是指这个方法可以接受任何字符串值（而不是强类型的变量）。当然，我们仍然需要知道类型名字的字符串形式！

Type.GetType()方法被重载，允许我们指定两个布尔类型的参数，一个用来控制当类型找不到时是否抛出异常，另一个用来指示是否区分字符串大小写。举例来说，考虑下面的情况：

```
// 使用静态的Type.GetType()方法获取类型信息（如果SportsCar没有找到，则忽略不抛出异常信息）
Type t = Type.GetType("CarLibrary.SportsCar", false, true);
```

在上面的例子中，注意传入GetType()的字符串没有包含类型所在的程序集信息。在这种情况下，该类型便被认为是定义在当前执行的程序集中的。但是，当希望得到一个外部私有程序集的类型元数据时，字符串参数必须使用类型完全限定名，加上类型所在程序集的友好名字（每一个都用逗号隔开）：

```
// 得到外部程序集中类型的类型信息
Type t = Type.GetType("CarLibrary.SportsCar, CarLibrary");
```

另外，传入Type.GetType()的字符串可以指定一个+标记来表示一个嵌套类型。如果希望得到一个嵌套在JamesBondCar类中的枚举类型（SpyOptions）的类型信息，可以写成下面这样：

```
// 得到当前程序集中嵌套枚举的类型信息
Type t =  Type.GetType("CarLibrary.JamesBondCar+SpyOptions");
```

15.3 构建自定义的元数据查看器

为了说明反射的基本过程（和System.Type的意义），建立一个名为MyTypeViewer的控制台程序。这个程序将显示mscorlib.dll（回想一下，所有.NET应用程序都自动访问这个核心框架类库）和MyTypeViewer中类型的方法、属性、字段和支持的接口（除一些指针外）的细节。创建好这个应用程序后，确保其中引入了System.Reflection命名空间。

```
// 需要为反射导入泛命名空间
using System.Reflection;
```

15.3.1 反射方法

修改Program类，定义一些静态方法，每个都带有一个System.Type参数并返回void类型。首先，编写ListMethods()方法，它（如读者猜想的那样）能够输出传入类型定义的每个方法的名称。注意Type.GetMethods()如何返回一个System.Reflection.MethodInfo类型的数组，它使用标准的foreach循环枚举，如下所示：

```
// 显示类型的方法名称
static void ListMethods(Type t)
{
  Console.WriteLine("***** Methods *****");
  MethodInfo[] mi = t.GetMethods();
  foreach(MethodInfo m in mi)
    Console.WriteLine("->{0}", m.Name);
```

```
  Console.WriteLine();
}
```

这里，只是使用MethodInfo.Name属性输出方法的名称。如你猜测的那样，MethodInfo有许多其他成员可以让你决定方法是否是静态的、虚拟的、泛型的或抽象的。此外，使用MethodInfo类型还能够获取方法的返回值和参数集。过一会儿我们再来整理ListMethods()的实现。

如果愿意，你可以建立一个合适的LINQ查询来枚举各个方法的名称。回忆第12章，LINQ to Object可以对内存中的对象集合构建强类型的查询。一般而言，只要有循环或条件编程逻辑，就可以使用相关的LINQ查询。例如，可以像下面这样修改上面的方法：

```
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    var methodNames = from n in t.GetMethods() select n.Name;
    foreach (var name in methodNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

15.3.2 反射字段和属性

ListFields()的实现和ListMethods()非常相似，不过值得注意的不同是：ListFields()调用了Type.GetFields()并且它的返回值是FieldInfo数组。此外，为了简明，我们使用LINQ查询只输出了每个字段的名称。

```
// 显示类型的字段名
static void ListFields(Type t)
{
  Console.WriteLine("***** Fields *****");
  var fieldNames = from f in t.GetFields() select f.Name;
  foreach (var name in fieldNames)
    Console.WriteLine("->{0}", name);
  Console.WriteLine();
}
```

显示类型属性的方法与此类似：

```
// 显示类型的属性名称
static void ListProps(Type t)
{
  Console.WriteLine("***** Properties *****");
  var propNames = from p in t.GetProperties() select p.Name;
  foreach (var name in propNames)
    Console.WriteLine("->{0}", name);
  Console.WriteLine();
}
```

15.3.3 反射实现的接口

接下来，我们将建立一个名为ListInterfaces()的方法，它将输出传入类型支持的所有接口名称。这里值得关注的是：对GetInterfaces()的调用返回一个System.Type类型的数组！这也可以理解，因为其实接口也是一种类型：

```
// 显示实现的接口
static void ListInterfaces(Type t)
{
  Console.WriteLine("***** Interfaces *****");
  var ifaces = from i in t.GetInterfaces() select i;
  foreach(Type i in ifaces)
    Console.WriteLine("->{0}", i.Name);
}
```

说明　注意，System.Type中的大多数get方法（GetMethods()、GetInterfaces()等）都包含指定BindingFlags枚举值的重载，可以对要搜索的内容进行较大的控制（如只搜索静态成员或只搜索公共成员和私有成员等）。详细内容请参考.NET Framework 4.5 SDK文档。

15.3.4　显示其他信息

最后，可以使用一个辅助方法来显示关于传入类型的各种统计信息（表示该类型是否是泛型，基类是什么，类型是否密封，等等）。

```
// 为了更好的检测
static void ListVariousStats(Type t)
{
  Console.WriteLine("***** Various Statistics *****");
  Console.WriteLine("Base class is: {0}", t.BaseType);
  Console.WriteLine("Is type abstract? {0}", t.IsAbstract);
  Console.WriteLine("Is type sealed? {0}", t.IsSealed);
  Console.WriteLine("Is type generic? {0}", t.IsGenericTypeDefinition);
  Console.WriteLine("Is type a class type? {0}", t.IsClass);
  Console.WriteLine();
}
```

15.3.5　实现Main()

在Program类中，Main()方法提示用户输入类型的完全限定名。一旦得到了这个字符串数据，就把它传入到Type.GetType()方法中，然后把（通过Type.GetType()）得到的System.Type类型再送到每个辅助方法中。这个过程可以不断重复，直到键入“Q”终止程序：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Welcome to MyTypeViewer *****");
  string typeName = "";

  do
  {
    Console.WriteLine("\nEnter a type name to evaluate");
    Console.Write("or enter Q to quit: ");

    // 得到类型的名称
    typeName = Console.ReadLine();

    // 询问用户是否想退出
    if (typeName.ToUpper() == "Q")
```

```
      {
        break;
      }

      // 测试显示类型
      try
      {
        Type t = Type.GetType(typeName);
        Console.WriteLine("");
        ListVariousStats(t);
        ListFields(t);
        ListProps(t);
        ListMethods(t);
        ListInterfaces(t);
      }
      catch
      {
        Console.WriteLine("Sorry, can't find type");
      }
    } while (true);
  }
```

现在，开始测试这个MyTypeViewer.exe。举例来说，运行应用程序并输入下面的完全限定名（注意Type.GetType()要求大小写完全匹配字符串）:

- System.Int32
- System.Collections.ArrayList
- System.Threading.Thread
- System.Void
- System.IO.BinaryWriter
- System.Math
- System.Console
- MyTypeViewer.Program

例如，下面显示了当指定System.Math时的部分输出。

```
***** Welcome to MyTypeViewer *****

Enter a type name to evaluate
or enter Q to quit: System.Math

***** Various Statistics *****
Base class is: System.Object
Is type abstract? True
Is type sealed? True
Is type generic? False
Is type a class type? True

***** Fields *****
->PI
->E

***** Properties *****

***** Methods *****
```

15

```
->Acos
->Asin
->Atan
->Atan2
->Ceiling
->Ceiling
->Cos
...
```

15.3.6　反射泛型类型

如果我们调用Type.GetType()来获取泛型类型的元数据描述，就必须使用包含“反勾号”(`)加上数字值的语法来表示类型支持的类型参数个数。例如，如果我们希望输出System.Collections.Generic.List<T>元数据描述，就需要为我们的应用程序传入如下字符串：

```
System.Collections.Generic.List`1
```

在这里，我们使用了数值1，这是因为List<T>只有一个类型参数。但如果希望反射DicTinary<TKey,TValue>，就需要提供值2，如下所示：

```
System.Collections.Generic.Dictionary`2
```

15.3.7　反射方法参数和返回值

到目前为止，一切顺利！接下来对当前应用程序做一个小的改进。我们需要特别修改ListMethods()辅助方法，使其不仅列出给定方法的名称，而且还列出方法的返回类型和输入参数类型。为此，MethodInfo类型提供了ReturnType属性和GetParameters()方法。在下面的代码中，注意，我们在用嵌套的foreach循环构造一个包括每个参数的类型和名称的字符串类型（没有使用LINQ）：

```
static void ListMethods(Type t)
{
  Console.WriteLine("***** Methods *****");
  MethodInfo[] mi = t.GetMethods();
  foreach (MethodInfo m in mi)
  {
    // 得到返回类型
    string retVal = m.ReturnType.FullName;
    string paramInfo = "( ";
    // 得到参数
    foreach (ParameterInfo pi in m.GetParameters())
    {
      paramInfo += string.Format("{0} {1} ", pi.ParameterType, pi.Name);
    }
    paramInfo += " )";

    // 现在显示基本方法
    Console.WriteLine("->{0} {1} {2}", retVal, m.Name, paramInfo);
  }
  Console.WriteLine();
}
```

如果现在运行修改后的程序，将对给定类型的方法有更多的了解。如果输入System.Object，将显

示下面的方法：

```
***** Methods *****
->System.String ToString (  )
->System.Boolean Equals ( System.Object obj  )
->System.Boolean Equals ( System.Object objA System.Object objB  )
->System.Boolean ReferenceEquals ( System.Object objA System.Object objB  )
->System.Int32 GetHashCode (  )
->System.Type GetType (  )
```

ListMethods()当前的实现很有用，因为我们可以直接使用System.Reflection对象模型来调查每一个参数和方法的返回类型。还可以有一个捷径，我们知道每一个XXXInfo类型（MethodInfo、PropertyInfo、EventInfo等）都重写了ToString()方法来显示请求项的签名。因此，我们也可以按如下所示实现ListMethods()方法（这里再次使用LINQ，其中只选择所有的MethodInfo方法，而不是Name值）：

```
static void ListMethods(Type t)
{
    Console.WriteLine("***** Methods *****");
    var methodNames = from n in t.GetMethods() select n;
    foreach (var name in methodNames)
        Console.WriteLine("->{0}", name);
    Console.WriteLine();
}
```

很有意思吧？很明显，System.Reflection命名空间和System.Type类允许我们反射的内容大大超过了MyTypeViewer当前显示的。如读者所希望的，我们可以获取类型的事件，得到某个成员的泛型参数列表以及其他的更多细节。

然而，对于当前创建的这个（还有些用的）对象浏览器，其主要的限制在于：仅仅能访问当前的程序集（MyTypeViewer）和总能访问的mscorlib.dll。故此，接下来的问题是："如何能使应用程序加载（并反射）在编译时并不知道的程序集？"

源代码 MyTypeViewer项目的源代码位于Chapter 15子目录下。

15.4 动态加载程序集

在第14章中，我们学习了CLR如何根据程序集清单来探测一个外部引用的程序集。虽然这很好，但是在很多时候，我们需要在运行时以编程的方式动态载入程序集，即使那些程序集没有记录在程序清单中。正式地说，这种按需加载外部程序集的操作被称为*动态加载*。

System.Reflection定义了一个名为Assembly的类。使用这个类，我们可以动态加载程序集，并找到关于程序集自身的属性。而且使用Assembly类型，我们还可以动态加载私有或共享程序集，还能够加载任意位置的程序集。从本质上说，Assembly类提供的方法（尤其是Load()和LoadFrom()）使你可以用编程的方式提供和客户端*.config文件中同样的信息。

为了说明动态加载，我们建立一个崭新的名为ExternalAssemblyReflector的控制台程序。我们的任

务是构造一个Main()方法，它将提示用户输入要动态加载的程序集友好名称。据此得到的Assembly引用将被传入名为DisplayTypes()的辅助方法，它将只输出所包含的每个类、接口、结构、枚举和委托的名称。实现代码非常简单：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.IO;  // 为文件找不到时抛出异常（FileNotFoundException）而定义

namespace ExternalAssemblyReflector
{
  class Program
  {
    static void DisplayTypesInAsm(Assembly asm)
    {
      Console.WriteLine("\n***** Types in Assembly *****");
      Console.WriteLine("->{0}", asm.FullName);
      Type[] types = asm.GetTypes();
      foreach (Type t in types)
        Console.WriteLine("Type: {0}", t);
      Console.WriteLine("");
    }

    static void Main(string[] args)
    {
      Console.WriteLine("***** External Assembly Viewer *****");

      string asmName = "";
      Assembly asm = null;

      do
      {
        Console.WriteLine("\nEnter an assembly to evaluate");
        Console.Write("or enter Q to quit: ");

        // 得到程序集名称
        asmName = Console.ReadLine();

        // 用户是否想退出
        if (asmName.ToUpper() == "Q")
        {
          break;
        }

        // 尝试加载程序集
        try
        {
          asm = Assembly.Load(asmName);
          DisplayTypesInAsm(asm);
        }
        catch
        {
          Console.WriteLine("Sorry, can't find assembly.");
        }
      } while (true);
```

```
    }
  }
}
```

注意，静态`Assembly.Load()`方法仅仅传入了一个要加载到内存的程序集的友好名称。因此，如果希望反射CarLibrary.dll，需要把CarLibrary.dll二进制文件复制到ExternalAssemblyReflector应用程序的\bin\Debug目录，然后再来运行这个程序。之后得到的输出结果类似于下面所示：

```
***** External Assembly Viewer *****

Enter an assembly to evaluate
or enter Q to quit: CarLibrary

***** Types in Assembly *****
->CarLibrary, Version=2.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9
Type: CarLibrary.MusicMedia
Type: CarLibrary.EngineState
Type: CarLibrary.Car
Type: CarLibrary.SportsCar
Type: CarLibrary.MiniVan
```

如果希望使ExternalAssemblyReflector更灵活，可以用`Assembly.LoadFrom()`而不是`Assembly.Load()`方法加载外部程序集，如下：

```
try
{
  asm = Assembly.LoadFrom(asmName);
  DisplayTypesInAsm(asm);
}
```

这样就可以输入要查看的程序集的绝对路径（如C:\MyApp\MyAsm.dll）。实质上，`Assembly.LoadFrom()`允许你以编程方式提供`<codeBase>`值。这样修改之后，你可以将一个完整路径传递给控制台应用程序。因此，如果CarLibrary.dll位于C:\MyCode下，可以得到如下输出结果：

```
***** External Assembly Viewer *****

Enter an assembly to evaluate
or enter Q to quit: C:\MyCode\CarLibrary.dll

***** Types in Assembly *****
->CarLibrary, Version=2.0.0.0, Culture=neutral, PublicKeyToken=33a2bc294331e8b9
Type: CarLibrary.EngineState
Type: CarLibrary.Car
Type: CarLibrary.SportsCar
Type: CarLibrary.MiniVan
```

15

源代码 ExternalAssemblyReflector项目的源代码位于Chapter 15子目录下。

15.5　反射共享程序集

Assembly.Load()方法被重载很多次了。Assembly.Load()的一种变化是允许指定一个区域设置（为本地化程序集）、一个版本号和公钥标记值（为共享程序集）。整体来说，识别一个程序集的一组术语称为显示名称（display name）。显示名称的格式以程序集友好名称开头，其后加上以逗号分隔的名称/值对字符串，后接可选的标识符（并可以按任意顺序出现）。下面是模板（可选项出现在括号中）：

```
Name (,Version = major.minor.build.revision) (,Culture = culture token)
(,PublicKeyToken= public key token)
```

在显示名称中，PublicKeyToken=null通常表示需要绑定和匹配一个非强名称的程序集。而Culture=""表示匹配目标机器默认的区域设置，举例来说：

```
// 使用默认的区域设置加载CarLibrary的1.0.0.0版本
Assembly a =
  Assembly.Load(@"CarLibrary, Version=1.0.0.0, PublicKeyToken=null, Culture=""");
```

另外，System.Reflection命名空间提供了AssemblyName类型，它允许用手工编写的对象变量来表示前面的字符串信息。通常，该类和System.Version（用面向对象封装了程序集的版本号）结合使用。以这种方式建立了显示名称后，就可以把它传入到重载的Assembly.Load()方法，如下所示：

```
// 使用AssemblyName定义显示名称
AssemblyName asmName;
asmName = new AssemblyName();
asmName.Name = "CarLibrary";
Version v = new Version("1.0.0.0");
asmName.Version = v;
Assembly a = Assembly.Load(asmName);
```

要加载一个GAC中的共享程序集，Assembly.Load()参数必须指定publikeytoken公钥标记值。举例来说，假定希望加载由.NET基础类库提供的System.Windows.Forms.dll程序集的4.0.0.0版本。由于此程序集中类型的数量非常大，所以下面的应用程序仅仅输出公有枚举的名称（通过简单的LINQ查询）：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;
using System.IO;

namespace SharedAsmReflector
{
  public class SharedAsmReflector
  {
    private static void DisplayInfo(Assembly a)
    {
      Console.WriteLine("***** Info about Assembly *****");
      Console.WriteLine("Loaded from GAC? {0}", a.GlobalAssemblyCache);
      Console.WriteLine("Asm Name: {0}", a.GetName().Name);
      Console.WriteLine("Asm Version: {0}", a.GetName().Version);
      Console.WriteLine("Asm Culture: {0}",
```

```
        a.GetName().CultureInfo.DisplayName);
      Console.WriteLine("\nHere are the public enums:");

      // 用LINQ查询找到公有枚举
      Type[] types = a.GetTypes();
      var publicEnums = from pe in types where pe.IsEnum &&
                        pe.IsPublic select pe;

      foreach (var pe in publicEnums)
      {
        Console.WriteLine(pe);
      }
    }

    static void Main(string[] args)
    {
      Console.WriteLine("***** The Shared Asm Reflector App *****\n");

      // 从GAC中加载System.Windows.Forms.dll
      string displayName = null;
      displayName = "System.Windows.Forms," +
        "Version=4.0.0.0," +
        "PublicKeyToken=b77a5c561934e089," +
        @"Culture=""";
      Assembly asm = Assembly.Load(displayName);
      DisplayInfo(asm);
      Console.WriteLine("Done!");
      Console.ReadLine();
    }
  }
}
```

源代码　SharedAsmReflector项目的源代码位于Chapter 15子目录下。

到这里，读者应当理解了如何在运行时使用System.Reflection命名空间中的核心成员来获得元数据。当然，虽然这很“酷”，但是在工作中可能不需要建立这样的自定义对象浏览器。尽管如此，反射服务仍然不可或缺，它是许多非常通用的编程开发的基础，包括晚期绑定。

15.6　晚期绑定

简单地说，晚期绑定（late binding）是一种创建一个给定类型的实例并在运行时调用其成员，而不需要在编译时知道它存在的一种技术。当建立一个晚期绑定到外部程序集类型的应用程序时，因为没有设置该程序集的引用，因此，调用程序清单没有直接列出这个程序集。

乍一看，晚期绑定的作用似乎不那么明显。如果可以“早期绑定”一个类型（比如，设定一个程序集引用并使用C# new关键字分配类型）的话，我们当然选择早期绑定。因为早期绑定能在编译时判断（类型）是否错误，而不是在运行时判断。但是晚期绑定对于程序的可扩展性来说至关重要。本章15.13节会构建可扩展应用程序，到那时，我们再看Activator类的作用。

15.6.1 System.Activator类

System.Activator类（定义在mscorlib.dll）是.NET晚期绑定过程中的关键所在。对于我们当前的例子，只需关注Activator.CreateInstance()方法，它用来建立一个晚期绑定类型的实例。为了适应多种情况，CreateInstance()方法经过了多次重载。其中最简单的变化是带有一个有效的Type对象，描述希望动态分配的实体。

为了说明该方法，接下来新建一个名为LateBindingApp的应用程序，使用C# using关键字导入System.IO和System.Reflection命名空间。现修改Program类如下：

```
// 这个程序将加载一个外部库并使用晚期绑定创建一个对象
public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Late Binding *****");
    // 尝试加载一个本地的CarLibrary副本
    Assembly a = null;
    try
    {
      a = Assembly.Load("CarLibrary");
    }
    catch(FileNotFoundException ex)
    {
      Console.WriteLine(ex.Message);
      return;
    }
    if(a != null)
      CreateUsingLateBinding(a);

    Console.ReadLine();
  }

  static void CreateUsingLateBinding(Assembly asm)
  {
    try
    {
    // 得到Minivan类型的元数据
      Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // 在运行时建立Minivan
      object obj = Activator.CreateInstance(miniVan);
      Console.WriteLine("Created a {0} using late binding!", obj);
    }
    catch(Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
}
```

在运行该应用之前，需要使用Windows Explorer将CarLibrary.dll手工复制到bin\Debug文件夹中。这是因为我们使用的是Assembly.Load()，因此CLR将只探测客户端文件夹（如果你愿意，可以使用Assembly.LoadFrom()向程序集输入一个路径，尽管这没有必要）。

说明 不要在本例中使用Visual Studio引用CarLibrary.dll。这会在客户端清单上记录该库。晚期绑定的要点是试图创建编译时未知的对象。

注意，Activator.CreateInstance()方法返回一个基本的System.Object类型，而不是一个强类型的MiniVan。因此，如果在obj变量上用点（.）操作，将不会看到任何MiniVan类的成员。乍一看，我们可以用显式强制类型转换解决这个问题：

```
// 是否需要通过强制转换来访问MiniVan的成员
// 不! 编译器错误
object obj = (MiniVan)Activator.CreateInstance(minivan);
```

但由于程序没有引用CarLibrary.dll，你不能使用C# using关键字引入CarLibrary命名空间，因此不能在转换操作中使用MiniVan。请记住，晚期绑定的重点是建立编译时未知对象的实例。因此，我们如何才能调用存储在System.Object引用中的MiniVan对象的底层方法呢？回答当然是使用反射。

15.6.2 调用没有参数的方法

假定希望调用MiniVan中的TurboBoost()方法。回想一下，这个方法设置引擎的状态为“dead”并显示一个消息框。第一步是使用Type.GetMethod()方法为TurboBoost()方法得到一个MethodInfo对象。接着，可以使用MethodInfo类型的Invoke()方法来调用MiniVan.TurboBoost。MethodInfo.Invoke()需要把所有的参数送到MethodInfo代表的方法中。这些参数用一组System.Object类型表示（作为方法的参数，可以是任意数量的不同对象实体）。

由于TurboBoost()不需要参数，所以只需传送null（意思是“这个方法没有参数”）。按如下代码所示修改CreateUsingLateBinding()方法：

```
static void CreateUsingLateBinding(Assembly asm)
{
  try
  {
    // 得到MiniVan类型的元数据
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // 在运行中建立MiniVan
    object obj = Activator.CreateInstance(miniVan);
    Console.WriteLine("Created a {0} using late binding!", obj);
    // 得到TurboBoost的信息
    MethodInfo mi = miniVan.GetMethod("TurboBoost");

    // 调用方法（'null'意味着没有参数）
    mi.Invoke(obj, null);
  }
  catch(Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

至此，一旦调用TurboBoost()方法，就可以看到如图15-2所示的消息框。

15

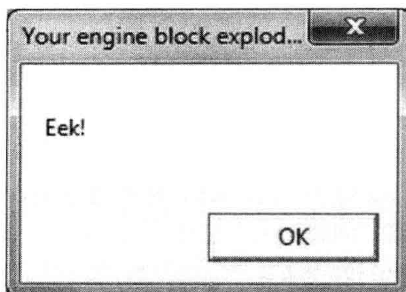


图15-2　晚期绑定方法调用

15.6.3　调用有参数的方法

在使用晚期绑定调用需要参数的方法时，要将参数打包到一个object类型的数组中。CarLibrary.dll 2.0.0.0版在Car类中定义了以下方法：

```
public void TurnOnRadio(bool musicOn, MusicMedia mm)
{
  if (musicOn)
    MessageBox.Show(string.Format("Jamming {0}", mm));
  else
    MessageBox.Show("Quiet time...");
}
```

该方法包含两个参数：布尔值表示汽车的音乐系统是否关闭，枚举值表示音乐播放器的类别。该枚举的结构如下：

```
public enum MusicMedia
{
  musicCd,      // 0
  musicTape,    // 1
  musicRadio,   // 2
  musicMp3      // 3
}
```

以下是Program类的新方法，它调用TurnOnRadio()。注意，我们使用MusicMedia枚举的数字值来表示“radio”媒体播放器：

```
static void InvokeMethodWithArgsUsingLateBinding(Assembly asm)
{
  try
  {
    // 首先，得到运动汽车的元数据描述
    Type sport = asm.GetType("CarLibrary.SportsCar");

    // 然后创建运动汽车
    object obj = Activator.CreateInstance(sport);

    // 调用包含参数的TurnOnRadio()
    MethodInfo mi = sport.GetMethod("TurnOnRadio");
    mi.Invoke(obj, new object[] { true, 2 });
  }
  catch (Exception ex)
```

```
    {
      Console.WriteLine(ex.Message);
    }
  }
```

这时，你可以看到反射、动态加载和晚期绑定之间的关系。当然，反射API所提供的特性远远不止这里提到的这些，如果感兴趣可以深入研究。

但你还是不清楚究竟何时可以在应用程序中使用这些技术。本章结束部分会解释这个问题，紧接下来我们先研究.NET特性的作用。

源代码 LateBindingApp项目的源代码位于Chapter 15子目录下。

15.7 .NET 特性的作用

本章开始已经说明，.NET编译器的任务之一是为所有定义和引用的类型生成元数据描述。除了程序集中标准的元数据外，.NET平台允许程序员使用特性（attribute）把更多的元数据嵌入到程序集中。简言之，特性就是用于类型（比如类、接口、结构等）、成员（比如属性、方法等）、程序集或模块的代码注解。

.NET特性是扩展了抽象的`System.Attribute`基类的类类型。当浏览.NET命名空间时，将发现许多预定义特性，可以在应用程序中使用它们。此外，可以创建自定义特性，通过从`Attribute`派生出新类型进一步修饰类型的行为。

.NET基础类库在不同的命名空间中提供了大量特性。表15-3给出了一些（绝对不是全部）预定义特性。

表15-3 预定义特性的少数几个例子

特　性	作　用
`[CLSCompliant]`	强制被注释项遵从CLS。前面已经说过，符合CLS的类型将确保无缝地跨越所有的.NET编程语言
`[DllImport]`	允许.NET代码调用任意非托管的C或C++基础类库，包括操作系统中的API。注意当与基于COM软件通信时，`[DllImport]`不能使用
`[Obsolete]`	标记一个不用的类或成员。如果其他程序员试图使用该项，他们将会收到一个描述出错信息的编译警告
`[Serializable]`	标记一个类或结构可以被“序列化”，意味着它可以将当前状态持久化到数据流中
`[NonSerialized]`	指定类或结构中的某个字段不能在序列化过程中被持久化
`[ServiceContract]`	标记一个方法是由WCF服务实现的契约

当在代码中应用特性时，如果它们没有被另一个软件显式地反射，那么嵌入的元数据基本没什么作用。反之，嵌入程序集中的元数据介绍将被忽略不计，而并无害处。

15.7.1　特性的使用者

与读者所猜想的一样，.NET Framework 4.5 SDK中的许多工具都需要查找各种特性。C#编译器（csc.exe）本身就要在编译周期中寻找各种特性是否存在。举例来说，如果C#编译器遇到[CLSCompilant]特性，它将自动检查特性项，确保它只公开符合CLS（公共语言规范的结构）。再举另外一个例子，如果C#编译器发现一个带有[Obsolete]特性的项，它将在Visual Studio的错误列表窗口中显示一个编译器警告。

除了开发工具，.NET基础类库中的许多方法也被设定为要反射指定的特性。举例来说，如果希望将一个对象的状态持久化到文件中，需要做的就是使用[Serializable]特性来注释类。如果Binary-Formatter类的Serialize()方法遇到这个特性，对象自动以紧凑的二进制形式被持久化到文件中。

最后，我们可以构建反射自定义特性和.NET基础类库中特性的应用程序。而这么做，本质上来说，是能够构建一组被特定的某些程序集理解的“关键字”。

15.7.2　在C#中使用特性

为了举例说明在C#中使用特性的过程，先创建一个名为ApplyingAttributes的新控制台应用程序。假定希望建立名为Motorcycle的类，它可以被持久化为二进制格式。要实现它，只需在类的定义中应用[Serializable]特性。如果有一个字段不想被持久化，可以应用[NonSerialized]特性：

```
// 该类可以保存到磁盘
 [Serializable]
public class Motorcycle
{
  // 可是这个字段不能被持久化
  [NonSerialized]
  float weightOfCurrentPassengers;

  // 这些字段要被持久化
  bool hasRadioSystem;
  bool hasHeadSet;
  bool hasSissyBar;
}
```

说明　一个特性只能被应用在紧接下来的对象。举例来说，在Motorcycle类中不能被序列化的字段仅是weightOfCurrentPassengers。而由于整个类中注释有[Serializable]，所以其他字段都可以被序列化。

这里，不考虑实际的对象序列化过程（第20章将学习详细内容）。我们只需要注意，当应用一个特性时，特性的名称被放在方括号中。

类被编译后，可以使用ildasm.exe查看更多的元数据。注意，这些被记录的特性使用了serializable（参见MotorCycle类中的红三角）和notserialized标记（在weightOfCurrentPassengers字段，如图15-3所示）。

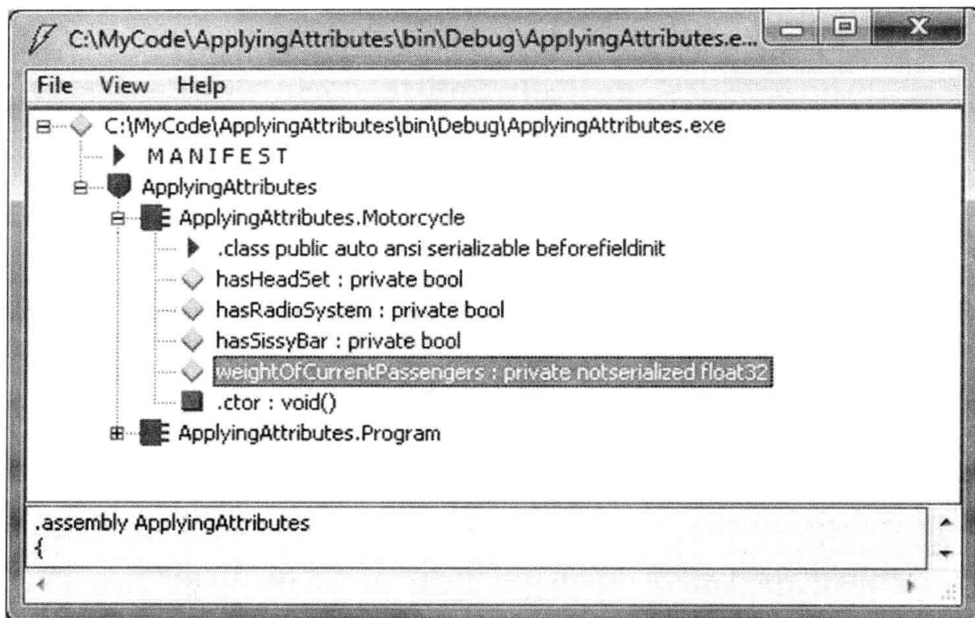


图15-3 在ildasm.exe中显示的特性

同读者猜想的一样，一个项可以被加上多种特性。假使有一个遗留的C#类类型（HorseAndBuggy）被标记为序列化的，但现在认为该类型对当前的开发来说已经过时。不必将类定义从代码库中删除（可能会破坏已有的软件），可以用[Obsolete]特性标记这个类。可以应用多个特性到一个单独项上，只需使用逗号分隔列出：

```
[Serializable, Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
  // ...
}
```

此外，也可以像下面一样在每个项上应用多个特性（最终结果是一样的）：

```
[Serializable]
[Obsolete("Use another vehicle!")]
public class HorseAndBuggy
{
  // ...
}
```

15.7.3 C#特性简化符号

如果仔细阅读.NET Framework 4.5 SDK文档，可能注意到带有[Obsolete]特性的实际类名是ObsoleteAttribute，而不是Obsolete。当名称转换时，所有.NET特性（包括自己建立的自定义特性）都将加上一个Attribute标记的后缀。但是，为简化应用特性的过程，C#语言不需要输入Attribute后缀。因此，下面HorseAndBuggy类型的迭代和先前是一样的（仅仅是多敲几个键而已）：

```
[SerializableAttribute]
[ObsoleteAttribute("Use another vehicle!")]
public class HorseAndBuggy
{
  // ...
}
```

注意，这是C#支持的，不是所有的.NET语言都支持这个特性。

15.7.4　为特性指定构造参数

注意，[Obsolete]特性可以接受一个构造参数。如果使用Code Definition窗口（可以使用Visual Studio的View菜单打开）查看[Obsolete]特性的正式定义，将发现这个类提供了一个构造函数，它接受一个System.String：

```
public sealed class ObsoleteAttribute : Attribute
{
  public ObsoleteAttribute(string message, bool error);
  public ObsoleteAttribute(string message);
  public ObsoleteAttribute();
  public bool IsError { get; }
  public string Message { get; }
}
```

当给特性提供构造参数时，直到该参数被其他类型或外部工具反射后，特性才被分配到内存中。定义在特性级的字符串数据只是作为元数据介绍被存储在程序集中。

15.7.5　Obsolete特性

既然HorseAndBuggy被标记了Obsolete特性，如果要使用该类的实例：

```
static void Main(string[] args)
{
  HorseAndBuggy mule = new HorseAndBuggy();
}
```

将发现提供的字符串数据被提取并显示在Visual Studio的Error List窗口中，并且当鼠标悬放在出错代码行的过期类型上时，也将显示该字符串数据（如图15-4所示）。

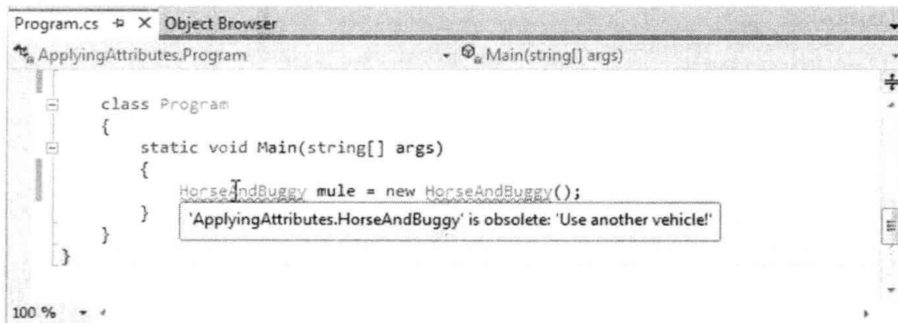


图15-4　特性

在本例中，反射[Obsolete]特性的“另一个软件”是C#编译器。希望此时你了解有关.NET特性的要点。

- 特性是派生自System.Attribute的类。
- 特性导致嵌入的元数据。
- 直到被其他代理反射，特性才发挥使用。
- 特性在C#中用方括号来应用。

接下来，让我们学习如何构建自定义特性，以及如何编写自定义的程序来反射嵌入的元数据。

源代码　ApplyingAttributes项目的源代码位于Chapter 15子目录下。

15.8　构建自定义特性

构建自定义特性的第一步是建立一个新的派生自System.Attribute的类。继续使用贯穿于本书的汽车的例子，假定构建了一个崭新的名为AttributedCarLibrary的C#类库，这个程序集定义了一些库，每一个都使用名为VehicleDescriptionAttribute的自定义特性来描述，如下所示：

```
// 自定义特性
public sealed class VehicleDescriptionAttribute : System.Attribute
{
  public string Description { get; set; }

  public VehicleDescriptionAttribute(string vehicalDescription)
  {
    Description = vehicalDescription;
  }
  public VehicleDescriptionAttribute(){ }
}
```

可见，VehicleDescriptionAttribute使用自动属性（Description）来维护一个字符串数据。该类除了派生自System.Attribute之外，其定义没有什么特别的。

说明　出于安全性的原因，考虑把所有的自定义特性都设计成密封的是.NET中一个好习惯。事实上，Visual Studio提供名为Attribute的代码段，它会将继承自System.Attribute的新类转储到代码窗口中。第2章完整第介绍了代码段的使用。记住，你可以输入代码段的名称然后按两次Tab键，这样就可以展开任何代码段。

15

15.8.1　应用自定义特性

考虑到VehicleDescriptionAttribute派生自System.Attribute，可以为汽车添加合适的注释。为了便于测试，在你的新类库中添加如下的类定义：

```
// 使用named property（命名属性）为description赋值
[Serializable]
```

```
[VehicleDescription(Description = "My rocking Harley")]
public class Motorcycle
{
}

[SerializableAttribute]
[ObsoleteAttribute("Use another vehicle!")]
[VehicleDescription("The old gray mare, she ain't what she used to be...")]
public class HorseAndBuggy
{
}

[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
}
```

15.8.2　命名属性语法

注意，Motorcycle的描述采用了一种新的描述方法，用到一个新的特性语法——*命名属性*（named property）。在第一个[VehicleDescription]特性的构造函数中，使用Description属性设置下面的字符串数据。如果该特性被一个外部代理反射，该值将写入到Description属性（只有特性提供一个可写的.NET属性，命名属性的语法才是合法的）。

与之相比，HorseAndBuggy和Winnebago类型没有使用命名属性的语法，而只是通过自定义构造函数传递了字符串数据。编译完AttributedCarLibrary程序集后，可以使用ildasm.exe查看在类型中放入的元数据描述。例如，图15-5显示了一个嵌入的Winnebago类的描述，特别是ildasm.exe中beforefieldinit中的数据。

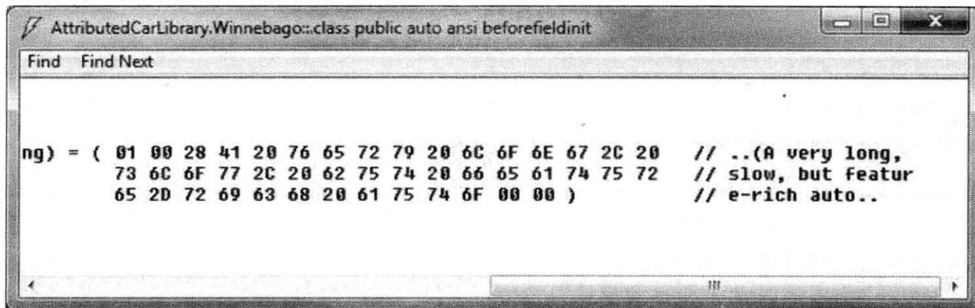


图15-5　嵌入的汽车描述数据

15.8.3　限制特性使用

默认情况下，自定义特性可以被应用在代码中几乎所有的方面（方法、类、属性等）。因此，只要恰当，可以使用VehicleDescription修饰方法、属性或字段（或其他的内容）：

```
[VehicleDescription("A very long, slow, but feature-rich auto")]
public class Winnebago
{
```

```
    [VehicleDescription("My rocking CD player")]
    public void PlayMusic(bool On)
    {
      ...
    }
  }
```

某些情况下，这样就能够满足需要。但是，有时候也许想建立这样一个自定义特性：它只被应用到选定的代码元素上。如果希望限制自定义特性的应用范围，需要在自定义特性的定义中应用[AttributeUsage]特性。[AttributeUsage]特性支持AttributeTargets枚举值的任意组合（通过OR操作）：

```
// 这个枚举定义了一个特性可能的目标值
public enum AttributeTargets
{
  All, Assembly, Class, Constructor,
  Delegate, Enum, Event, Field, GenericParameter,
  Interface, Method, Module, Parameter,
  Property, ReturnValue, Struct
}
```

此外，[AttributeUsage]也允许我们随意设置命名属性，比如AllowMultiple，它用来指示在相同项上特性是否可被应用多次（默认值为false）。而[AttributeUsage]也允许我们使用Inherited命名属性指示特性是否能够被派生类继承（默认值为true）。

为了设定[VehicleDescription]特性只能在类或结构中应用一次，可以修改VehicleDescription-Attribute定义如下：

```
// 这次，我们使用AttributeUsage特性来注释我们的自定义特性
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
                Inherited = false)]
public sealed class VehicleDescriptionAttribute : System.Attribute
{
...
}
```

基于此，如果开发者试图应用[VehicleDescription]特性到类或结构以外的其他地方，就会遇到编译时错误。

15.9 程序集级别特性

使用和[assembly:]标签，在给定程序集的所有类型上应用特性也是可以的。举例来说，假定你希望确保定义在程序集中的每个公共类型都是符合CLS（公共语言规范）的。

说明 第1章提到了遵循CLS的程序集的作用。遵循CLS的程序集可用于所有.NET编程语言。如果公共类型的公共成员公开了未遵循CLS的编程结构（如无符号数据或指针参数），其他.NET语言将无法使用这些功能。因此，如果要构建可用于各种.NET语言的C#代码库，则必须遵循CLS。

要实现这个目标，只需在每个C#源代码文件顶部中加入下面的程序集级别特性。注意程序集或者模块级别特性必须在命名空间范围外定义！如果向项目添加程序集（或模块）级别特性，推荐的文件布局如下：

```
// 首先列出using语句
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

// 现在列出所有程序集/模块级别特性
// 强制所有在程序集中的公共类型符合CLS
 [assembly: CLSCompliant(true)]

// 下面列出命名空间和类型
namespace AttributedCarLibrary
{
  // 类
}
```

如果现在增加不符合CLS的代码（如未标记数据的公开点）：

```
// Ulong类型不符合CLS
public class Winnebago
{
  public ulong notCompliant;
}
```

就会收到一个编译器错误。

Visual Studio AssemblyInfo.cs文件

默认情况下，Visual Studio生成一个名为AssemblyInfo.cs的文件（如图15-6所示）。可以通过扩展Solution Explorer的Properties图标来查看它。

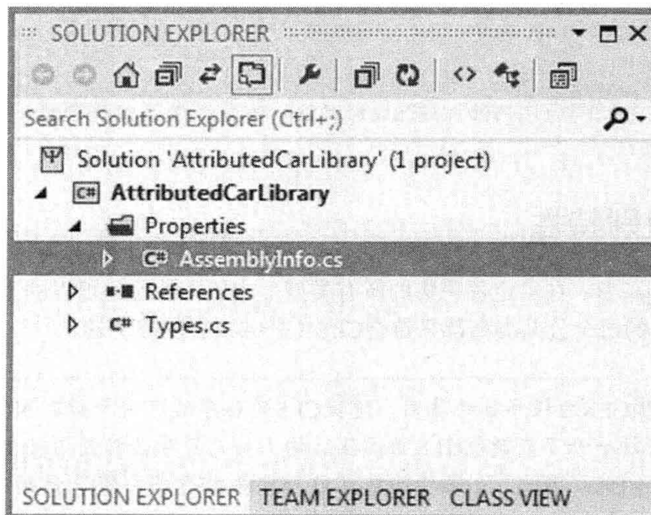


图15-6　AssemblyInfo.cs文件

这个文件用于放置程序集级别使用的特性。第14章讨论过.NET程序集，其中的清单包含程序集级别的元数据，表15-4列出了一些需要注意的程序集级别的特性。

表15-4 部分程序集级别特性

特性	含义
[AssemblyCompany]	保存基本的公司信息
[AssemblyCopyright]	保存产品或程序集的版权信息
[AssemblyCulture]	提供程序集支持的区域或语言信息
[AssemblyDescription]	保存组成程序集的产品或模块的描述
[AssemblyKeyFile]	指定包含用于签名程序集的密钥对的文件的名称（例如建立一个强名称）
[AssemblyProduct]	提供产品信息
[AssemblyTrademark]	提供商标信息
[AssemblyVersion]	指定程序集的版本信息，用<major.minor.build. revision>格式

源代码 AttributedCarLibrary项目的源代码位于Chapter 15子目录下。

15.10 使用早期绑定反射特性

记住，一个特性直到另一个软件反射它的值时才有用。给定的特性被发现后，软件可以采取任何需要的行为。现在，和应用程序一样，这个“另一个软件”也可以使用早期或晚期绑定找到一个自定义特性。如果希望使用早期绑定，则相应的特性需要客户应用程序在编译时定义（在本例中是VehicleDescriptionAttribute）。既然AttributedCarLibrary程序集把这个自定义特性定义为公用类，早期绑定就是最好的选择。

为说明反射自定义特性的过程，创建一个新的C#控制台程序，命名为VehicleDescriptionAttributeReader。然后，设置AttributedCarLibrary程序集的引用。最后，用下面代码修改初始*.cs文件：

```
// 使用早期绑定反射自定义特性
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using AttributedCarLibrary;

namespace VehicleDescriptionAttributeReader
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
      ReflectOnAttributesUsingEarlyBinding();
      Console.ReadLine();
    }

    private static void ReflectOnAttributesUsingEarlyBinding()
    {
      // 得到一个表现Winnebago的类型
```

```
        Type t = typeof(Winnebago);

        // 得到Winnebago所有的特性
        object[] customAtts = t.GetCustomAttributes(false);

        // 输出描述
        foreach (VehicleDescriptionAttribute v in customAtts)
          Console.WriteLine("-> {0}\n", v.Description);
      }
    }
  }
```

Type.GetCustmeAttributes()方法返回一个对象数组，表示了应用到Type代表的成员上的所有特性（Type是布尔类型的参数，控制是否扩展搜索到继承链）。得到特性列表后，遍历每个Vehicle-DescriptionAttribute类并输出Description属性的值。

源代码　VehicleDescriptionAttributeReader应用程序的源代码位于Chapter 15子目录下。

15.11　使用晚期绑定反射特性

前面的例子使用早期绑定输出Winnebago类型汽车的描述数据，这是因为VehicleDescription-Attribute类类型在AttributedCarLibrary程序集中被定义为公共成员。也可以使用动态加载和晚期绑定来反射特性。

创建新的名为VehicleDescriptionAttributeReaderLateBinding的项目，将AttributedCarLibrary.dll复制到项目的\bin\Debug目录下，现在，修改Program类如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Reflection;

namespace VehicleDescriptionAttributeReaderLateBinding
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Value of VehicleDescriptionAttribute *****\n");
      ReflectAttributesUsingLateBinding();
      Console.ReadLine();
    }

    private static void ReflectAttributesUsingLateBinding()
    {
      try
      {
        // 加载本地的AttributedCarLibrary的副本
        Assembly asm = Assembly.Load("AttributedCarLibrary");

        // 得到VehicleDescriptionAttribute的类型信息
```

```
        Type vehicleDesc =
          asm.GetType("AttributedCarLibrary.VehicleDescriptionAttribute");

        // 得到Description属性的类型信息
        PropertyInfo propDesc = vehicleDesc.GetProperty("Description");

        // 得到程序集中所有类型
        Type[] types = asm.GetTypes();

        // 遍历每个类型，得到所有的VehicleDescriptionAttributes
        foreach (Type t in types)
        {
          object[] objs = t.GetCustomAttributes(vehicleDesc, false);

          // 遍历每个VehicleDescriptionAttribute并使用晚期绑定输出描述
          foreach (object o in objs)
          {
            Console.WriteLine("-> {0}: {1}\n",
            t.Name, propDesc.GetValue(o, null));
          }
        }
      }
      catch (Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
  }
}
```

如果读者一直在看本章中的例子程序，这段代码应该（多多少少）不需加以说明了。仅需要关注的是PropertyInfo.GetValue()方法的使用，它用来触发属性的访问者。下面显示了当前示例的输出结果：

```
***** Value of VehicleDescriptionAttribute *****

-> Motorcycle: My rocking Harley

-> HorseAndBuggy: The old gray mare, she ain't what she used to be...

-> Winnebago: A very long, slow, but feature-rich auto
```

源代码 VehicleDescriptionAttributeReaderLateBinding应用程序的源代码位于Chapter 15子目录下。

15.12 反射、晚期绑定和自定义特性的使用背景

尽管已经运行了这些技术的多个示例，读者可能仍然想知道什么时候在程序中使用反射、动态加载、晚期绑定和自定义特性。的确，这些有趣的主题似乎有一点学院派的味道（有的读者可能认为这不是什么坏事）。为帮助将这些主题应用到实际情况中，需要一个实际的例子。假定现在读者在一个编程团队中，该团队需要按照下面的需求建立一个应用程序。

15

- 产品必须可以通过使用第三方工具进行扩展。

然而，什么是可扩展的呢？想想Visual Studio IDE。当应用程序开发时，各种“钩子”被插入，允许其他软件提供商在IDE开发环境中插入自定义模块。显然，Visual Studio团队没有办法设置引用还未编写的外部.NET程序集（因此，不能使用早期绑定），所以，如何让应用程序提供需要的钩子呢？下面是解决这个问题的一个可能的办法。

- 首先，可扩展的应用程序必须提供一些输入手段，允许用户指定被插入的模块（比如一个对话框或命令行标志）。这需要*动态加载*。
- 其次，为了插入到环境中，可扩展的应用程序必须要确定模块是否支持正确的功能（比如一组需要的接口）。这需要*反射*。
- 最后，可扩展的应用程序必须获取一个需要的基础架构的引用（例如接口类型）并调用成员触发底层功能。这经常需要*晚期绑定*。

简单来说，如果可扩展的应用程序预编程为查询指定的接口，则它可以在运行时确定类型是否可以被激活。一旦验证测试通过，类型便可以支持额外的接口，为它们的功能提供多种结构。这正是Visual Studio团队采取的做法，我觉得一点都不困难。

15.13　构建可扩展的应用程序

下面通过一个完整的例子说明构建一个可扩展的Windows窗体应用程序的过程，它可以使用外部程序集来扩展功能。如果没有用Windows Forms API构建过GUI，你应该加载提供的解决方案代码，然后继续。

说明　Windows Forms是.NET平台最初的桌面API。但从.NET 3.0开始，WPF API迅速成为GUI框架的首选。尽管如此，本书仍将使用Windows Forms作为一些客户端GUI的示例，因为它的代码与相应WPF代码相比，要稍显直观。

如果不熟悉建立Windows窗体应用程序的过程，只需要打开提供的示例代码，然后按照下面的步骤来做。具体说明一下，我们的可扩展应用程序需要下列程序集。

- CommonSnappableTypes.dll：该程序集包含将被每个插件对象实现的类型定义，该类型定义将会被Windows窗体应用程序直接引用。
- CSharpSnapIn.dll：一个用C#编写的插件，它使用CommonSnappableTypes.dll类型。
- VbSnapIn.dll：一个用Visual Basic编写的插件，它使用CommonSnappableTypes.dll类型。
- MyExtendableApp.exe：这个Windows窗体应用程序将成为可以被每个插件功能扩展的实体。

另外，这个应用程序可以使用动态加载、反射和晚期绑定来动态获取预先不知道的程序集的功能。

15.13.1　构建CommonSnappableTypes.dll

第一步是建立一个程序集，它将包含能将插件插入可扩展Windows窗体应用程序中的类型。`CommonSnappableTypes`类库项目定义了两个类型：

```
namespace CommonSnappableTypes
{
  public interface IAppFunctionality
  {
    void DoIt();
  }

  [AttributeUsage(AttributeTargets.Class)]
  public sealed class CompanyInfoAttribute : System.Attribute
  {
    public string CompanyName { get; set; }
    public string CompanyUrl { get; set; }
  }
}
```

IAppFunctionality接口为可被可扩展Windows窗体应用程序使用的所有插件提供了一个多态接口。当然，这纯粹是举例，所以我们只提供单个DoIt()方法。为了接近实际的例子，想象有这样一个接口（或一组接口），它（们）可以允许插件产生脚本代码，可以在应用程序工具箱中生成图像或整合到承载程序的主菜单中。

CompanyInfoAttribute类型是一个自定义特性，可以用于希望插到容器中的任意类类型。通过该类的定义可以知道，[CompanyInfo]允许插件的开发者提供一些关于组件来源的基本细节。

15.13.2 构建C#插件

接下来，需要建立一个实现IAppFunctionality接口的类型。当然，为集中精力做可扩展应用程序的总体设计，我们只用一个小的类型。假定新的C#代码库项目命名为CSharpSnapIn，它定义了一个CSharpModule类型。该类必须使用CommonSnappableTypes中定义的类型，因此，请保证设置了对这个二进制文件的引用（另外，用于消息提示的System.Windows.Forms.dll也要引用进来）。说了这么多，这里是代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using CommonSnappableTypes;
using System.Windows.Forms;

namespace CSharpSnapIn
{
  [CompanyInfo(CompanyName = "My Company",
      CompanyUrl = "www.MyCompany.com")]
  public class CSharpModule : IAppFunctionality
  {
    void IAppFunctionality.DoIt()
    {
      MessageBox.Show("You have just used the C# snap in!");
    }
  }
}
```


注意，当支持IAppFunctionality接口时，选择了显式的接口实现。虽然这并不是必需的，但这里主要考虑到系统中需要与该接口类型直接交互的部分只有承载的Windows程序。通过显式实现接口，DoIt()方法没有直接从CSharpModule类型中公开。

15.13.3　构建Visual Basic插件

有些第三方厂商更喜欢用Visual Basic而不用C#，为模拟它们的角色，构建一个新的Visual Basic类库（VbNetSnapIn），与前一个CSharpSnapIn项目引用相同的外部程序集。

说明　默认情况下，Visual Basic项目在Solution Explorer中不显示References文件夹。要在VB项目中添加引用，可以使用Visual Studio中的Project →Add Reference...菜单选项。

代码（再次）有意写得比较简单：

```
Imports System.Windows.Forms
Imports CommonSnappableTypes

<CompanyInfo(CompanyName:="Chucky's Software", CompanyUrl:="www.ChuckySoft.com")>
Public Class VbSnapIn
  Implements IAppFunctionality

  Public Sub DoIt() Implements CommonSnappableTypes.IAppFunctionality.DoIt
    MessageBox.Show("You have just used the VB snap in!")
  End Sub
End Class
```

要注意的是，Visual Basic中特性的应用是使用尖括号（< >）而不是方括号（[ ]）。还要注意，`Implements`关键字用来在指定的类或结构上实现接口类型。

15.13.4　构建可扩展的Windows Forms应用程序

最后一步是建立一个新的Windows Forms应用程序（MyExtendableApp），它能够让用户使用标准的Windows Open对话框选择一个插件程序。如果你以前没创建过Windows Forms应用程序，可以选择Visual Studio中New Project对话框中的Windows Forms Application项目（如图15-7所示）。

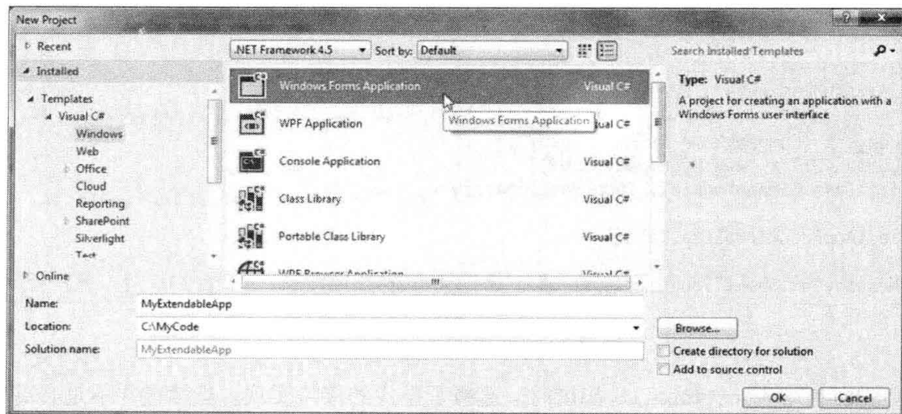


图15-7　使用Visual Studio创建Windows Forms项目

现在，添加CommonSnappableTypes.dll的引用，不用添加CSharpSnapIn.dll或VbSnapIn.dll的引用。同时，在窗体的主代码文件（右击窗体设计器，选择View Code，可以打开主代码文件）中引入System.Reflection和CommonSnappableTypes命名空间。记住，该应用程序的唯一目标，就是利用晚期绑定和反射来决定由第三方厂商所创建的独立二进制文件的“可插取性”。

我不想在这里说明Windows Forms开发的细节（参见附录A）。但是假定在窗体模板中放置了一个MenuStrip组件，定义一个置顶的菜单项，命名为File，它提供一个名为Snap In Module的子菜单项。该Windows窗体也将包含ListBox类型（这里被重命名为lstLoadedSnapIns），该类型用于显示由用户加载的每一个插件程序的名称。图15-8显示了最终的GUI界面。

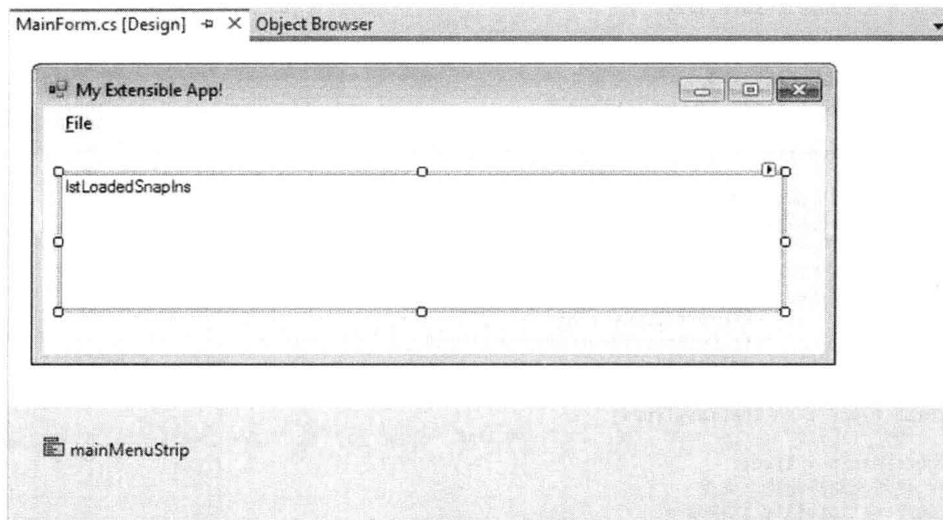


图15-8　MyExtendableApp的GUI界面

处理File Snap In Module菜单项的Click事件的代码（在设计时编辑器中双击菜单项即可创建）显示一个FileOpen对话框，并且取出所选文件的路径。假定用户没有选择CommonSnappableTypes.dll程序集（因为这完全是基础结构），这个路径传递到一个名为LoadExternalModule()的辅助方法中来处理。当不能找到实现IAppFunctionality的类时，这个方法将返回false：

```
private void snapInModuleToolStripMenuItem_Click(object sender,
  EventArgs e)
{
  // 允许用户选择一个程序集加载
  OpenFileDialog dlg = new OpenFileDialog();
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    if(dlg.FileName.Contains("CommonSnappableTypes"))
      MessageBox.Show("CommonSnappableTypes has no snap-ins!");
    else if(!LoadExternalModule(dlg.FileName))
      MessageBox.Show("Nothing implements IAppFunctionality!");
  }
}
```

LoadExternalModule()方法执行下列任务：

- ❑ 将程序集动态加载到内存中；
- ❑ 确定程序集是否包含实现IAppFunctionality的类型；
- ❑ 用晚期绑定实现类型。

如果发现一个实现IAppFunctionality的类型，DoIt()方法将被调用，并且类型的完全限定名将被添加到ListBox中（注意foreach循环将遍历程序集中的所有类型，以应对一个单一程序集拥有多个插件的可能性）。

```
private bool LoadExternalModule(string path)
{
  bool foundSnapIn = false;
  Assembly theSnapInAsm = null;
  try
  {
    // 动态加载选中的程序集
    theSnapInAsm = Assembly.LoadFrom(path);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
    return foundSnapIn;
  }
  // 得到程序集中所有的IAppFunctionality兼容的类
  var theClassTypes = from t in theSnapInAsm.GetTypes()
                      where t.IsClass &&
                      (t.GetInterface("IAppFunctionality") != null)
                      select t;
  // 创建对象对调用DoIt()方法
  foreach (Type t in theClassTypes)
  {
    foundSnapIn = true;
    // 使用晚期绑定建立类型
    IAppFunctionality itfApp =
      (IAppFunctionality)theSnapInAsm.CreateInstance(t.FullName, true);
    itfApp.DoIt();
    lstLoadedSnapIns.Items.Add(t.FullName);
  }
  return foundSnapIn;
}
```

到这里就可以运行应用程序了。当选择CSharpSnapIn.dll或VbSnapIn.dll程序集时，将看到正确的信息显示。最后的任务是显示被[CompanyInfo]特性支持的元数据。为此，只需修改LoadExternalModule()方法，在退出foreach语句块前调用一个名为DisplayCompanyData()的新辅助方法。注意该方法带有一个System.Type的参数。

```
private bool LoadExternalModule(string path)
{
...
  foreach (Type t in theClassTypes)
  {
...
    // 显示公司信息
    DisplayCompanyData(t);
  }
  return foundSnapIn;
}
```

使用传入的类型，反射得到［CompangInfo］特性，如下所示：

```
private void DisplayCompanyData(Type t)
{
  // 获取 [CompanyInfo] 数据
  var compInfo = from ci in t.GetCustomAttributes(false) where
                 (ci.GetType() == typeof(CompanyInfoAttribute))
                 select ci;
  // 显示数据
  foreach (CompanyInfoAttribute c in compInfo)
  {
    MessageBox.Show(c.CompanyUrl,
      string.Format("More info about {0} can be found at", c.CompanyName));
  }
}
```

图15-9显示了可能的运行情况。

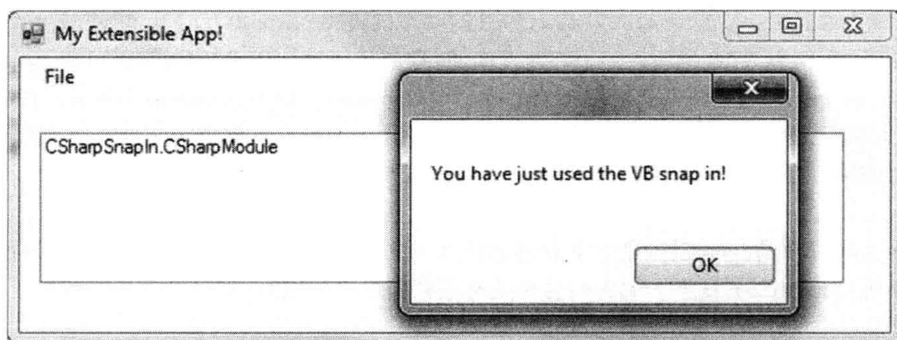


图15-9　插入外部程序集

太棒了，我们完成了这个应用程序示例。希望到这里读者可以看到本章表现的主题在实际环境中非常有用，并且对构建工具没有限制。

源代码 CommonSnappableTypes、CSharpSnapIn、VbSnapIn和MyExtendableApp项目的源代码位于Chapter 15子目录下。

15.14　小结

反射是健壮的面向对象环境中非常有意思的特性。在.NET中，反射服务与System.Type类和System.Reflection命名空间紧密相关。我们已经看到，反射在运行时把一个类型放置在放大镜下，理解一个给定项是谁、做什么、在哪里、什么时候、为什么做和怎样做的过程。

晚期绑定是创建类型并调用其成员的过程，它无须预先知道成员的特定名称。晚期绑定是动态加载的直接结果，它允许通过编程方式在内存中加载.NET程序集。在本章的可扩展应用程序示例中可以看到，这是在工具构建和使用中非常有用的技术。本章还学习了基于特性编程的作用。使用特性修饰类型，结果是增加了底层程序集元数据。

15

第 16 章 动态类型和动态语言运行时

16

微软在.NET 4.0中为C#引入了一个新的关键字dynamic，它允许我们在类型安全的分号和花括号之间的强类型世界里使用脚本化的行为。使用这种松散的类型，可以极大地简化一些复杂的编码任务，而且还可以获得与大量基于.NET的动态语言（如IronRuby、IronPython）交互的能力。

本章，我们将学习C# dynamic关键字的方方面面，理解如何使用DLR①（Dynamic Language Runtime，动态语言运行时）将松散的类型映射到正确的内存对象。理解了DLR提供的诸多服务之后，你将看到一些示例，它们使用动态类型来简化后期绑定方法的调用（通过反射服务）并且可以方便地与遗留的COM库进行通信。

说明 不要混淆C#中的dynamic关键字和动态程序集（见第18章）的概念。虽然在创建动态程序集时可以使用dynamic关键字，但它们其实是两个完全不同的概念。

16.1 dynamic 关键字的作用

在第3章中我们学习了var关键字。使用var可以定义本地变量，该变量的实际数据类型取决于编译时，是在初次分配时确定的（称为隐式类型）。在初次分配之后，你将拥有一个强类型的变量，任何不相容的赋值操作都会导致编译错误。

在开始深入研究C#的dynamic关键字之前，先创建一个名为DynamicKeyword的控制台应用程序。接下来，在Program类中创建下面的方法，并确保如果没有注释掉最后一行语句将会触发一个编译时错误。

```
static void ImplicitlyTypedVariable()
{
  // a为List<int>类型
  var a = new List<int>();
  a.Add(90);

  // 下面这一行将导致编译时错误
  // a = "Hello";
}
```

---

① DLR是以.NET为基础的诸如IronPython和IronRuby之类的动态语言核心。它提供了一个环境，可以很容易地实现动态语言，以及添加动态能力到C#这样的静态类型语言。——编者注

仅仅为了使用隐式类型而使用隐式类型被很多人认为是一个糟糕的编程风格（如果你确定需要一个List<int>，那就声明一个List<int>吧）。但是在第13章中你已经看到，隐式类型对于LINQ来说是非常有用的，因为很多LINQ查询都会返回（经过投影的）匿名类型的枚举，而这种匿名类是无法在C#代码中直接声明的。但即使在这种情况下，这些隐式类型变量实际上也是强类型的。

同样，在第6章中我们了解到System.Object是.NET Framework的顶级父类，可以代表任何类型。那么如果声明了一个object类型的变量，就会得到一个强类型的数据，尽管它所指向的内存区域会因引用的分配而有所不同。为了访问内存中该对象引用所指向的成员，需要进行显示转换。

假设有一个Person类，它包含两个类型为string的自动属性（FirstName和LastName）。如下面的代码所示：

```
static void UseObjectVarible()
{
  // 假设已经存在一个名为Person的类
  object o = new Person() { FirstName = "Mike", LastName = "Larson" };

  // 要访问Person的属性，必须将object转换为Person
  Console.WriteLine("Person's first name is {0}", ((Person)o).FirstName);
}
```

随着.NET 4.0的发布，C#语言引入一个关键字dynamic。从高层看，我们可以把任何值设置为动态的数据类型，因此可以认为dynamic关键字是一个特殊形式的System.Object。乍看上去，这似乎很容易混淆，因为我们已经有了三种定义数据的方法，这些数据的实际类型不是直接在代码库中指定的。例如下面的方法：

```
static void PrintThreeStrings()
{
  var s1 = "Greetings";
  object s2 = "From";
  dynamic s3 = "Minneapolis";

  Console.WriteLine("s1 is of type: {0}", s1.GetType());
  Console.WriteLine("s2 is of type: {0}", s2.GetType());
  Console.WriteLine("s3 is of type: {0}", s3.GetType());
}
```

将打印出如下的内容：

```
s1 is of type: System.String
s2 is of type: System.String
s3 is of type: System.String
```

动态类型之所以与隐式声明的类型或者通过System.Object引用声明的类型有着巨大的不同，是因为动态类型不是强类型的。或者说，动态数据不是静态类型（statically typed）。对于C#编译器来说，通过dynamic关键字声明的数据点可以分配任意初始值，而且可以在其生命周期内重新分配任何新的（甚至可能无关的）值。来看下面的代码及其输出结果：

16

```
static void ChangeDynamicDataType()
{
  // 声明一个名为t的动态数据点
  dynamic t = "Hello!";
```

```
  Console.WriteLine("t is of type: {0}", t.GetType());

  t = false;
  Console.WriteLine("t is of type: {0}", t.GetType());

  t = new List<int>();
  Console.WriteLine("t is of type: {0}", t.GetType());
}
```

```
t is of type: System.String
t is of type: System.Boolean
t is of type: System.Collections.Generic.List`1[System.Int32]
```

在学习过程中你会发现，上面的代码如果把变量t声明为System.Object，那么编译过程和执行结果会完全相同。但是一会儿你就会察觉，dynamic关键字提供了很多其他特性。

16.1.1　调用动态声明的数据的成员

由于动态数据类型可以在运行时代表任何类型（就像一个System.Object类型的变量），那么接下来你可能要问如何调用动态变量的成员（属性、方法、索引器、注册事件，等等）。其实，在语法上它们仍然没有什么区别。你只需要在动态数据变量后面加一个点（“.”），然后指定其公共成员，并在必要时提供相应的参数。

但是，编译器将不会检查你所指定的成员的有效性！记住，与System.Object定义的变量不同，动态数据不属于静态类型。直到运行时你才会知道所调用的动态数据是否支持指定的成员、所传递的参数是否正确以及成员的拼写是否准确无误，等等。因此，下面的代码可以通过编译，尽管它看上去可能很奇怪：

```
static void InvokeMembersOnDynamicData()
{
  dynamic textData1 = "Hello";
  Console.WriteLine(textData1.ToUpper());

  // 你可能认为下面的代码会有编译错误，但实际上它们能通过编译
  Console.WriteLine(textData1.toupper());
  Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
}
```

你可能会注意到第二次调用WriteLine()时程序试图调用动态数据点的toupper()方法。如你所见，textData1为string类型，并不包含所有字母均为小写的同名方法。并且，string也不包含名为Foo()参数分别为int、string、DateTime类型的方法！

但是这已经符合了C#编译器的要求。只不过在Main()方法中调用该方法时，将会得到如下所示的运行时错误：

```
未处理的异常：Microsoft.CSharp.RuntimeBinder.RuntimeBinderException：
'string'不包含名为'toupper'的定义。
```

调用动态数据的成员与调用强类型数据的成员之时，另一个明显的差别是，如果在动态数据之后输入“.”操作符，Visual Studio将不会显示期望的智能感知，而是得到如图16-1所示的提示。

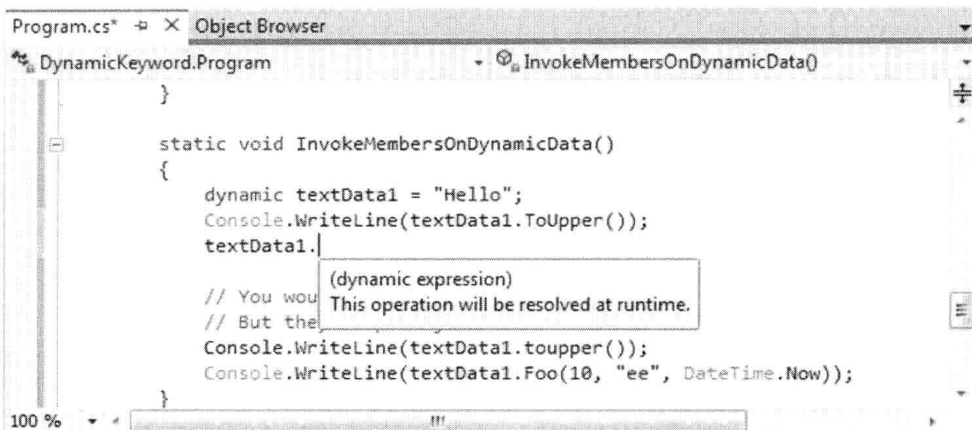


图16-1 动态数据不会激活智能感知

动态数据无法使用智能感知是可以理解的。但这也意味着在编写包含这种数据点的C#代码时需要格外小心。任何成员的拼写或大小写错误都会抛出运行时错误，通常是`RuntimeBinderException`类的一个实例。

16.1.2 Microsoft.CSharp.dll程序集的作用

当使用Visual Studio新建一个C#项目时，将自动引用一个名为Microsoft.CSharp.dll的程序集（可以在Solution Explorer的References文件夹下找到）。这个库非常小，只定义了一个命名空间（`Microsoft.CSharp.RuntimeBinder`）和两个类（如图16-2所示）。

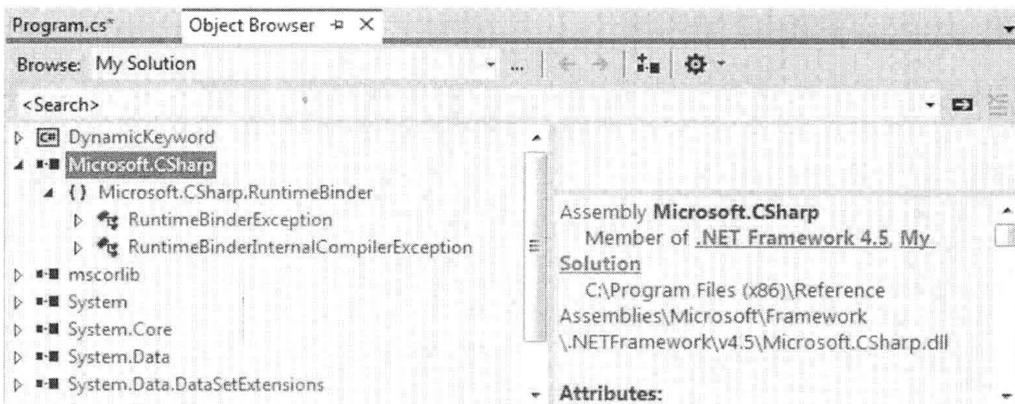


图16-2 Microsoft.CSharp.dll程序集

顾名思义，这两个类为强类型的异常类。RuntimeBinderException是最普通的类，如果试图调用一个不存在的动态数据类型的成员，将会抛出该异常（比如调用toupper()和Foo()方法）。如果调用了一个存在的成员但却指定了错误的参数数据，将同样会抛出该异常。

由于动态数据的这种不确定性，在调用用C#的dynamic关键字声明的变量的成员时，可以用合适的try/catch块来包裹，并以一种优雅的方式来处理异常。如下所示：

```
static void InvokeMembersOnDynamicData()
{
  dynamic textData1 = "Hello";

  try
  {
    Console.WriteLine(textData1.ToUpper());
    Console.WriteLine(textData1.toupper());
    Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
  }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

在调用这个方法时，会发现对ToUpper()方法（注意T和U这两个字母）的调用正确无误，但控制台上还是会显示一条错误数据。

```
HELLO
'string' does not contain a definition for 'toupper'
```

当然，如果将所有动态方法的调用都用try/catch块来包裹，这个过程将是十分麻烦的。只要注意了拼写和参数传递，就没有必要都进行包裹。但是，当你不确定目标类型是否包含某个成员的时候，如果用try/catch块来进行包裹的话，那么异常的捕获就会十分方便了。

16.1.3　dynamic关键字的作用域

（用var关键字声明的）隐式类型数据只能作为一个成员范围内的本地变量。var关键字不能用于返回值、参数或类/结构的成员。但对于dynamic关键字来说，这都不是问题。来看下面这个类的定义：

```
class VeryDynamicClass
{
  // 动态字段
  private static dynamic myDynamicField;

  // 动态属性
  public dynamic DynamicProperty { get; set; }

  // 动态返回值类型和动态参数类型
  public dynamic DynamicMethod(dynamic dynamicParam)
  {
    // 动态本地变量
    dynamic dynamicLocalVar = "Local variable";

    int myInt = 10;
```

```
        if (dynamicParam is int)
        {
          return dynamicLocalVar;
        }
        else
        {
          return myInt;
        }
      }
    }
```

现在你可以如愿调用这些公共成员了，但由于所操作的是动态方法和属性，因此并不能完全确定它们的数据类型！可以肯定，这个VeryDynamicClass类在实际的应用程序中用处不大，但它确实演示了dynamic关键字的作用域。

16.1.4 dynamic关键字的限制

虽然dynamic关键字可以定义很多东西，但它使用起来也存在一些限制。尽管瑕不掩瑜，但你仍然要清楚的是，在调用一个动态数据的方法时，不能使用Lambda表达式和C#匿名方法。例如，即使目标方法的参数确实是一个值为string并返回void的委托，但下面的代码也总是会报错。

```
dynamic a = GetDynamicObject();

// 错误！动态数据的方法不能使用Lambda表达式
a.Method(arg => Console.WriteLine(arg));
```

要避免这个限制，可以直接使用基本的委托，详见第10章中描述的各种技术（匿名方法、Lambda表达式等）。另一个限制是动态数据点无法理解扩展方法（见第11章）。不幸的是，LINQ API所提供的所有扩展方法也都包含在内。因此，用dynamic关键字声明的变量不能用于LINQ to Object以及其他LINQ技术：

```
dynamic a = GetDynamicObject();

//错误！无法找到动态数据的Select()扩展方法
var data = from d in a select d;
```

16.1.5 dynamic关键字的实际用途

动态数据不是强类型的，无法进行编译时检查，不能触发智能感知并且无法进行LINQ查询，因此，认为为了使用dynamic关键字而使用它是非常差的编程实践一点也没有错。

但是在某些场景中，dynamic关键字可以显著地减少手工输入的代码量。特别是在构建一个需要大量使用后期绑定（通过反射）的.NET应用程序时，dynamic关键字可以节省大量打字时间。同样，如果构建一个需要与遗留的COM库（如微软Office产品）进行交互的.NET应用程序，可以使用dynamic关键字极大地简化代码库。

和其他“捷径”一样，你需要权衡利弊。使用dynamic关键字是在用类型的安全性来换取代码的简洁度。由于C#本质上还是一门强类型的语言，因此你需要分析应用场景来决定是否使用动态行为。要记住dynamic关键字并不是必需的。你总是可以手工编写替换代码（通常要远远多于使用dynamic关键字的代码）来得到同样的结果。

源代码　DynamicKeyword项目的源代码位于Chapter 16子目录下。

16.2 DLR 的作用

既然已经理解了什么是"动态数据"，接下来我们将学习如何处理这种动态数据。DLR随.NET 4.0一起发布，它是作为CLR（Common Language Runtime，公共语言运行时）的补充的运行时环境。"动态运行库"（dynamic runtime）当然不是新概念。事实上很多编程语言，如Smalltalk、LISP、Ruby和Python，已经使用这个概念很多年了。简单地说，动态运行时允许动态语言完全在运行时发现类型，而不进行编译时检查。

如果你有强类型语言（包括不含动态类型的C#语言）的背景知识，这种运行时的概念可能看上去不太受欢迎。毕竟在任何可能的地方，你只希望得到编译时错误，而不是运行时错误。不过动态语言/运行时确实提供了下面一些有趣的特性。

- 极其灵活的代码库。在重构时不需要频繁修改数据类型。
- 在不同平台和编程语言所创建的对象类型之间进行互操作非常简便。
- 可以在运行时为内存中的类型添加或移除成员。

DLR的一个作用是使不同的动态语言能够在.NET运行时运行，并提供一种与其他.NET代码进行交互的方式。两个最受欢迎的使用DLR的动态语言是IronPython和IronRuby。这两种语言属于动态领域，它们的类型只有在运行时才能被发现。不过它们都可以使用.NET丰富的基础类库。更妙的是，由于`dynamic`关键字的引入，它们的代码库可以与C#进行互操作（反之亦然）。

说明　本章不会介绍如何用DLR与动态语言集成。详细内容请参考IronPython（http://ironpython.codeplex.com）和IronRuby（http://rubyforge.org/projects/ironruby）网站。

16.2.1 表达式树的作用

DLR在非特定条件下使用表达式树来获取动态调用的含义。例如，当DLR遇到如下的C#代码时：

```
dynamic d = GetSomeData();
d.SuperMethod(12);
```

它将自动创建一个表达式树，"调用对象d的SuperMethod方法，并将数字12作为参数"。然后，该信息（正式的名称为有效载荷）被传递给正确的运行时绑定器（binder），可能仍然是C#动态绑定器，也可能为IronPyton动态绑定器，甚至遗留的COM对象（稍后会进行解释）。

这时，请求被映射到目标对象所需的调用结构。表达式树（实际上我们并不需要手动创建它们）所带来的好处是，我们可以编写固定的C#语句，而不必担心潜在的目标究竟是什么（COM对象、IronPython和IronRuby代码库，等等）。图16-3从较高的层次演示了表达式树的概念。

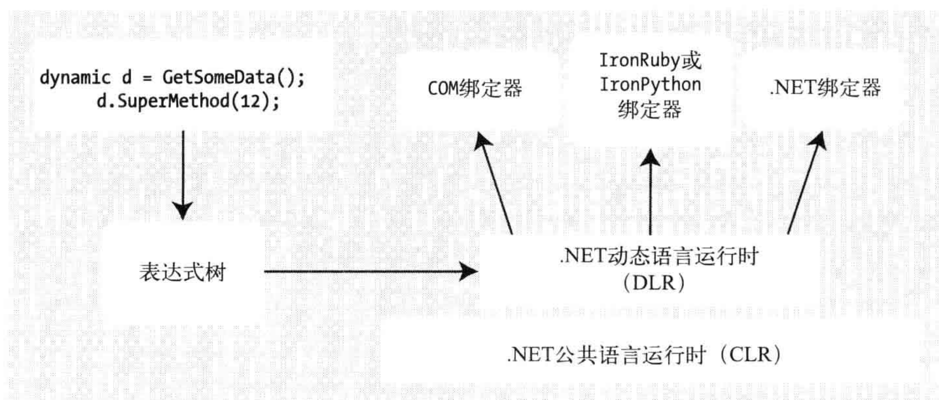


图16-3 表达式树在非特定条件下获取动态调用并通过绑定器进行处理

16.2.2 System.Dynamic命名空间的作用

System.Dynamic命名空间放置于System.Core.dll程序集中。说实话，我们几乎不需要使用该命名空间中的类型。但如果作为一个语言供应商，希望让自己的动态语言与DLR进行交互，那么就需要使用System.Dynamic命名空间来创建一个自定义的运行时绑定器。

同样也不必深究本书提到的System.Dynamic中的类型，但如果感兴趣可以随时查询.NET Framewrok 4.5 SDK文档。出于实际的目的，只需要简单地知道该命名空间提供了必要的结构，使得一门.NET语言具备动态的特性。

16.2.3 表达式树的动态运行时查找

如前所述，DLR将表达式树传递给目标对象。但这种指派会受到一些因素的影响。如果动态数据类型指向一个内存中的COM对象，那么表达式树将被发送给一个低级别的COM接口IDispatch。正如你可能知道的那样，该接口以COM的方式合并了它本身的动态服务集。虽然你可以在.NET应用程序中不通过DLR和C#的dynamic关键字而使用COM对象，但这往往会导致更加复杂的C#代码（稍后将会看到）。

如果动态数据指向的不是COM对象，那么表达式树将可能传递给实现了IDynamicObject接口的对象。该接口在后台使用，它支持类似IronRuby那样的语言接收一个DLR表达式树，并将其映射到Ruby特性（Ruby Specifics）。

最后，如果动态数据既不指向COM对象，也不指向实现IDynamicObject接口的对象，而是指向一个普通的.NET对象，那么表达式树会指派给C#运行时绑定器来进行处理，并且会将其映射到.NET的细节。这个映射过程包含了反射服务。

表达式树交给一个给定的绑定器进行处理后，当包含必要参数的方法被正确地调用之后，动态数据将被解析为真正的内存中的数据类型。现在，让我们来看一些DLR的实际应用，就从简化后期绑定的.NET调用开始吧。

16.3 使用动态类型简化后期绑定调用

在使用反射服务时，你也许会决定使用dynamic关键字，特别在是进行后期绑定的方法调用时。在第15章中，我们举了一些例子介绍了使用这种非常有用的方法的最佳时机是在创建扩展的应用程序时。在第15章我们学习了如何使用Activator.CreateInstance()方法创建一个object，除了它的显示名称之外，我们不知道任何有关这个对象的编译时信息。接下来可以使用System.Reflection命名空间下的类型通过后期绑定来调用其成员。回忆一下第15章中的这个例子：

```
static void CreateUsingLateBinding(Assembly asm)
{
  try
  {
    // 获取Minivan类型的元数据
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // 在运行时创建Minivan类型
    object obj = Activator.CreateInstance(miniVan);

    // 获取TurboBoost的信息
    MethodInfo mi = miniVan.GetMethod("TurboBoost");

    // 调用方法（'null'代表没有参数）
    mi.Invoke(obj, null);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

虽然这段代码运行良好，但却不得不承认它有一点笨拙。我们必须手动使用MethodInfo类、手动查询元数据，等等。现在我们使用C#的dynamic关键字和DLR来重写这个方法：

```
static void InvokeMethodWithDynamicKeyword(Assembly asm)
{
  try
  {
    // 获取Minivan类型的元数据
    Type miniVan = asm.GetType("CarLibrary.MiniVan");

    // 在运行时创建Minivan并调用其方法
    dynamic obj = Activator.CreateInstance(miniVan);
    obj.TurboBoost();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

使用dynamic关键字来声明obj变量，可以用DLR优雅地实现烦琐的反射操作。

利用dynamic关键字传递参数

当你需要通过后期绑定调用包含参数的方法时，DLR的好处就更加明显了。使用“冗长”的反射调用时，需要将所有参数打包到一个object数组中，然后将这个数组传递给MethodInfo的Invoke()方法。

为了用一个全新的示例来进行演示，我们先来创建一个C#控制台应用程序LateBinding-WithDynamic。然后在当前解决方案下添加一个类库项目MathLibrary（使用File→Add New Project...菜单项）。将MathLibrary项目初始的Class1.cs文件重命名为SimpleMath.cs，并用如下的代码来实现：

```
public class SimpleMath
{
  public int Add(int x, int y)
  {
    return x + y;
  }
}
```

在编译MathLibrary.dll程序集时，将其复制一份放到LateBindingWithDynamic项目的\bin\Debug文件夹下（也可以单击Solution Explorer下各个项目的Show All Files按钮，然后在项目之间拖曳文件）。这时，Solution Explorer将如图16-4所示。

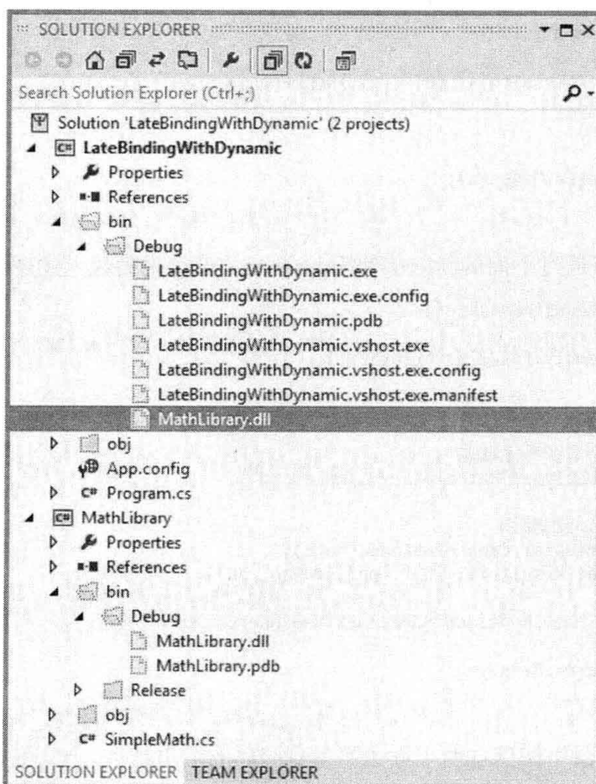


图16-4 LateBindingWithDynamic项目包含一个私有的MathLibrary.dll副本

说明　记住，后期绑定的关键在于允许应用程序在没有某个对象的清单数据（manifest）的情况下创建该对象。因此我们将MathLibrary.dll手动复制到控制台项目的输出文件夹下，而没有使用Visual Studio添加该程序集的引用。

现在，在控制台应用程序项目的Program.cs文件中引入System.Reflection命名空间。然后在Program类中添加如下的方法。该方法使用典型的反射API调用Add()方法。

```
private static void AddWithReflection()
{
  Assembly asm = Assembly.Load("MathLibrary");

  try
  {
    // 获取SimpleMath类型的元数据
    Type math = asm.GetType("MathLibrary.SimpleMath");

    // 在运行时创建SimpleMath
    object obj = Activator.CreateInstance(math);

    // 获取Add方法的信息
    MethodInfo mi = math.GetMethod("Add");

    // 调用方法（包含其参数）
    object[] args = { 10, 70 };
    Console.WriteLine("Result is: {0}", mi.Invoke(obj, args));
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

现在，思考一下如何通过下面的新方法使用dynamic关键字来简化上述逻辑：

```
private static void AddWithDynamic()
{
  Assembly asm = Assembly.Load("MathLibrary");

  try
  {
    // 获取SimpleMath类型的元数据
    Type math = asm.GetType("MathLibrary.SimpleMath");

    // 在运行时创建SimpleMath
    dynamic obj = Activator.CreateInstance(math);
    Console.WriteLine("Result is: {0}", obj.Add(10, 70));
  }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

不错吧！在Main()方法中调用这两个方法将得到同样的输出结果。但是使用dynamic关键字可以节省相当多的工作量。使用动态定义的数据时，你不必手动将参数打包成对象数组，也不必手动查询程序集元数据等其他类似的细节。如果你正在构建大量使用动态加载和后期绑定的应用程序，你肯定会

发现，随着时间的推移，这种节省代码量的效果会越来越明显。

源代码　LateBindingWithDynamic项目的源代码位于Chapter 16子目录下。

16.4　使用动态数据简化 COM 互操作

现在我们来看看dynamic关键字在一个与COM互操作的项目中所发挥的巨大作用。如果你没有丰富的COM开发背景，那么只需要知道本例中编译好的COM库和其他.NET库一样包含元数据，只不过它们的格式是完全不同的。因此，如果某个.NET程序要与COM对象通信，第一个步骤就是生成一个大家所熟悉的“互操作程序集”（稍后会介绍）。这一步是非常简单的，只需要激活Add Reference对话框，选择COM选项卡找到要使用的COM库（如图16-5所示）。

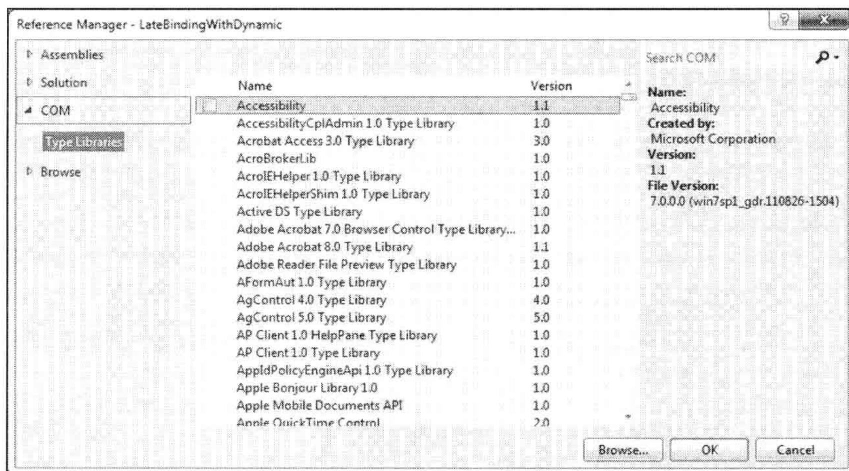


图16-5　Add Reference对话框的COM选项卡显示了电脑中所有已注册的COM库

说明　要知道一些重要的微软对象模型（包括Office产品）现在已经可以通过COM互操作来访问了。因此，即使你没有直接构建COM应用程序的经验，也可以在.NET程序中使用它们。

选择了一个COM库后，IDE将为我们生成一个全新的程序集，该程序集包含COM元数据的.NET描述。其正式的名称为互操作程序集（interoperability assembly，或简写为interop assembly）。互操作程序集中除了有一小部分将COM事件转换为.NET事件的代码之外，不包含任何实现代码。互操作程序集使.NET代码库避免了内部COM的复杂性，因此是非常有用的。

由于CLR（如果使用dynamic关键字的话就是DLR）可以自动对.NET数据类型和COM类型进行映射，因此可以直接在C#代码中使用互操作程序集。在后台，使用Runtime Callable Wrapper（RCW）对数据进行封送，使其在.NET和COM应用程序之间交互。从根本上说，RCW是一个动态生成的代理，

它对.NET数据类型进行封装并转换为COM类型，将COM的返回值映射为等价的.NET类型。图16-6显示了.NET与COM进行交互的全景图。

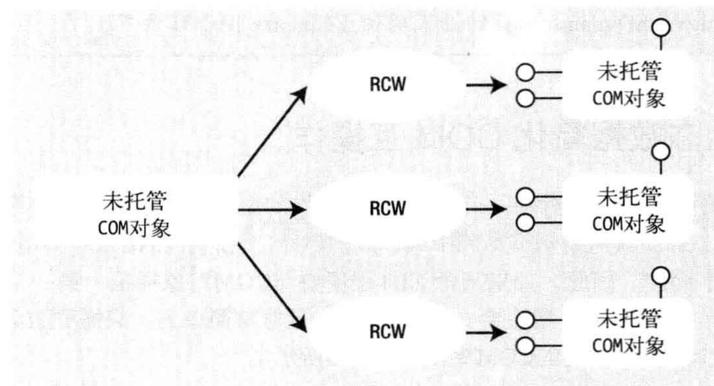


图 16-6　使用 RCW 代理使.NET 程序与 COM 对象交互

16.4.1　主互操作程序集的作用

COM库供应商创建的很多COM库（如可以访问微软Office产品的对象模型的微软COM库）都提供了一个“官方”的互操作程序集，称为*主互操作程序集*，简称PIA。PIA是优化的互操作程序集。使用Add Reference对话框添加COM库的引用时会生成一些代码，PIA要比这些代码更整洁，也更具扩展性。

PIA与核心的.NET库一样存在于Add Reference对话框的Assemblies的列表中（位于Extensions子区域）。事实上，如果在Add Reference对话框的COM选项卡中引用了一个COM库，Visual Studio不会像往常那样生成一个新的互操作库，而是会使用提供的PIA。图16-7显示了微软Office Excel对象模型的PIA，我们将在下一个示例中用到它。

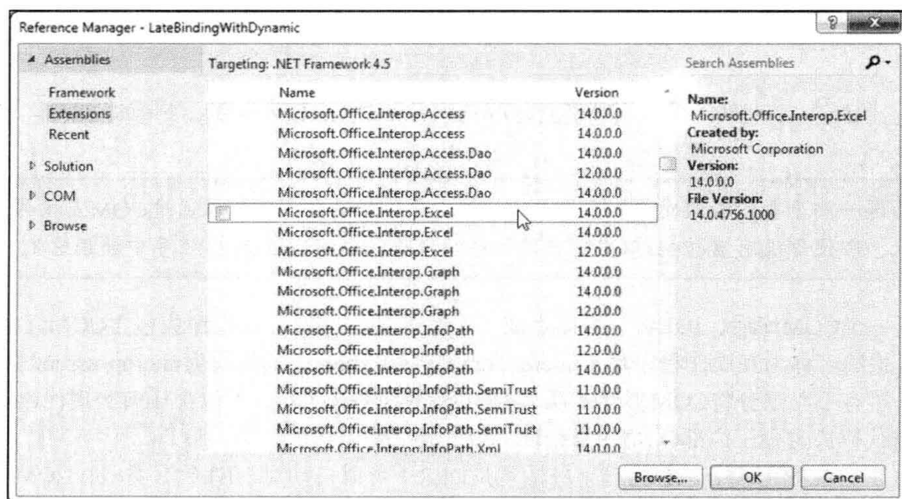


图16-7　PIA存在于Add Reference对话框.NET选项卡的列表中

16.4.2　嵌入互操作元数据

在.NET 4.0发布之前，如果C#应用程序要使用COM库（PIA或其他），我们需要确保客户端计算机上包含该互操作程序集的副本。这不但会增加应用程序安装包的尺寸，而且安装脚本还必须检查PIA程序集是否存在，如果不存在还要在GAC中安装副本。

然而在.NET 4.0或更高版本下，你可以选择将互操作数据直接嵌入到编译的.NET应用程序中。这样，必要的互操作元数据已经硬编码到.NET程序中，我们就不必在.NET应用程序中携带互操作程序集的副本了。

默认情况下，当使用Add Reference对话框选择一个COM库（PIA或其他）时，IDE会自动将该库的Embed Interop Types属性设置为True。在Solution Explorer的References文件夹中选择一个引用的互操作库，然后打开其Properties窗口，在该窗口中可以第一时间看到这项设置（如图16-8所示）。

C#编译器将仅包含互操作程序集中你所真正使用的那部分。因此，如果互操作库包含许多COM对象的.NET描述，你所能得到的仅仅是在C#代码里真正使用的那部分子集的定义。这样做除了能减小发布到客户端的应用程序的规模，而且由于不必在目标机器上安装所有缺失的PIA，还能简化安装路径。



图16-8　互操作程序集逻辑可以直接嵌入到.NET应用程序中

16.4.3 普通COM互操作的难点

在下一个示例之前，我们先来介绍另一个预备话题。在DLR发布之前，在编写需要使用COM库（通过互操作程序集）的C#代码时，你肯定会面对各种各样的挑战。例如，很多COM库定义的方法使用了可选参数，.NET 3.5之前的C#不支持。这就需要为每个可选参数指定Type. Missing值。比如一个COM方法包含5个参数，并且都是可选的，那么你需要编写如下的C#代码来接受默认的值：

```
myComObj.SomeMethod(Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

幸运的是，如果不指定特殊的值，则可以编写如下所示的简化代码，Type.Missing值将在编译时插入。

```
myComObj.SomeMethod();
```

此外，很多COM方法还提供了对命名参数的支持。正如第4章中所提到的那样，它允许以任意顺序向成员传递参数。由于C支持了这个特性，因此可以“跳过”我们不关心的那些可选参数，而只设置我们关心的参数。

COM互操作的另一个普遍难点是，很多COM方法的参数或返回值都是非常特殊的数据类型，我们称之为Variant。与C#的dynamic关键字类似，可以在运行时将Variant数据类型指定为任意的COM数据类型（字符串、接口、引用、数值等）。在dynamic关键字诞生之前，要想传递和接收Variant数据点需要一些迂回方法，比如大量的转换操作。

如果将Embed Interop Types属性设置为True，所有的COM Variant类型都将自动映射为dynamic数据。这不仅避免了在使用COM Variant数据类型时进行大量无关的转换操作，而且隐藏了COM的一些复杂性，如COM索引。

现在需要创建一个使用微软Office对象模型的应用程序，以展示如何使用C#的可选参数、命名参数以及dynamic关键字来简化COM互操作。在整个示例中，我们将使用这些新特性，并将其工作量与不使用时进行对比。

说明 如果你没有Windows Forms的背景，可以简单地在Visual Studio中加载完整的解决方案，并实验这些代码。不要手工构建应用程序。

16.5 使用 C# 动态数据进行 COM 互操作

假设我们有一个名为ExportDataToOfficeApp的Windows Form GUI应用程序，主窗体中承载了一个名为dataGridCars的DataGridView控件。窗体中还有两个Button控件，其中一个可以打开自定义对话框向列表中插入一条新的数据，另一个可以将列表中的数据导入到一个Excel表格中。图16-9显示了完整的GUI。

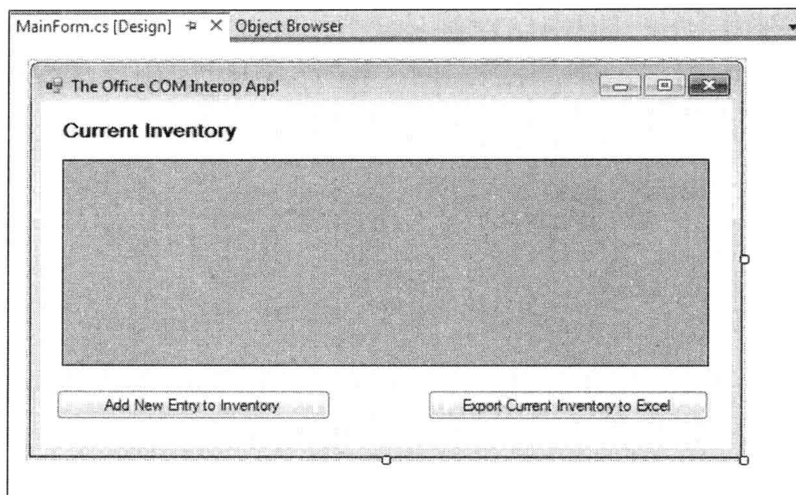


图16-9 COM互操作示例的GUI

我们通过处理窗体的Load事件来给DataGridView控件填充一些初始数据（项目中的Car类包含Color、Make、PetName属性，它将作为List<T>泛型类的类型参数）：

```
public partial class MainForm : Form
{
  List<Car> carsInStock = null;

  public MainForm()
  {
    InitializeComponent();
  }

  private void MainForm_Load(object sender, EventArgs e)
  {
    carsInStock = new List<Car>
    {
      new Car {Color="Green", Make="VW", PetName="Mary"},
      new Car {Color="Red", Make="Saab", PetName="Mel"},
      new Car {Color="Black", Make="Ford", PetName="Hank"},
      new Car {Color="Yellow", Make="BMW", PetName="Davie"}
    };

    UpdateGrid();
  }

  private void UpdateGrid()
  {
    // 重新设置数据源
    dataGridCars.DataSource = null;
    dataGridCars.DataSource = carsInStock;
  }
}
```

“Add New Entry to Inventory”按钮的Click事件将打开一个自定义的对话框，用户可以输入一个Car对象的新数据，然后单击OK按钮，数据将被添加到列表中（我不打算给出对话框的后台代码，有关

细节请查看所提供的解决方案）并将NewCarDialog.cs、NewCarDialog.designer.cs、NewCarDialog. rexs文件添加到项目中（所有这些文件都是代码的一部分）。完成上述工作后，为“Add”按钮在主窗体的click处理程序提供如下实现：

```
private void btnAddNewCar_Click(object sender, EventArgs e)
{
  NewCarDialog d = new NewCarDialog();
  if (d.ShowDialog() == DialogResult.OK)
  {
    // 向列表中添加一个新的Car
    carsInStock.Add(d.theCar);
    UpdateGrid();
  }
}
```

“Export Current Inventory to Excel”按钮的Click事件处理程序是该示例的核心部分。打开Add Reference对话框，添加对Microsoft.Office.Interop.Excel.dll主互操作程序集的引用（如前面的图16-7所示）。在窗体的主代码文件中添加如下的命名空间别名。请注意定义别名并不是与COM库交互时所必需的，但这么做可以为所有引用的COM对象提供一个方便的限定符。当某些COM对象的名称与你的.NET类型名称发生冲突时，这么做是十分便利的解决方法。

```
// 为Excel对象模型创建一个别名
using Excel = Microsoft.Office.Interop.Excel;
```

“Export”按钮的Click事件处理程序调用了一个私有的辅助函数ExportToExcel()：

```
private void btnExportToExcel_Click(object sender, EventArgs e)
{
  ExportToExcel(carsInStock);
}
```

由于我们使用Visual Studio引用COM库，所以PIA将自动配置，所需的元数据将嵌入到.NET应用程序中（Embed Interop Types属性的作用）。因此，所有的COM Variant将被识别为动态数据类型。此外，我们还可以使用C#可选参数和命名参数。考虑如下的ExportToExcel()实现：

```
static void ExportToExcel(List<Car> carsInStock)
{
  // 加载Excel，新建一个空的工作簿
  Excel.Application excelApp = new Excel.Application();
  excelApp.Workbooks.Add();

  // 本示例需要一个工作表
  Excel._Worksheet workSheet = excelApp.ActiveSheet;

  // 在单元格中创建列头
  workSheet.Cells[1, "A"] = "Make";
  workSheet.Cells[1, "B"] = "Color";
  workSheet.Cells[1, "C"] = "Pet Name";

  // 现在，将所有List<Car>中的数据映射到工作表中
  int row = 1;
  foreach (Car c in carsInStock)
  {
    row++;
    workSheet.Cells[row, "A"] = c.Make;
    workSheet.Cells[row, "B"] = c.Color;
```

```
    workSheet.Cells[row, "C"] = c.PetName;
  }

  // 美化表数据
  workSheet.Range["A1"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

  // 保存文件，退出Excel并将信息显示给用户
  workSheet.SaveAs(string.Format(@"{0}\Inventory.xlsx", Environment.CurrentDirectory));
  excelApp.Quit();
  MessageBox.Show("The Inventory.xslx file has been saved to your app folder",
    "Export complete!");
}
```

该方法将Excel加载到内存中，因此在电脑桌面上是看不到文件的。对于该应用程序来说，我们只对如何使用内部Excel对象模型感兴趣。不过，如果你真的希望显示Excel的UI，可以在方法中添加一行代码：

```
static void ExportToExcel(List<Car> carsInStock)
{
  // 加载Excel，新建一个空的工作簿
  Excel.Application excelApp = new Excel.Application();

  // 使Excel在计算机中可见
  excelApp.Visible = true;

...
}
```

在创建一个新的工作表时，我们添加了3个列，它们的名字与Car类的属性名相同。在这之后，使用List<Car>中的数据填充单元格，然后将文件保存为（硬编码）Inventory.xlsx。

这时，如果运行程序，添加新的记录，并将数据导出到Excel，就可以打开Inventory.xlsx文件了，它保存在Windows Form应用程序的\bin\Debug文件夹下。图16-10显示了导出的结果。

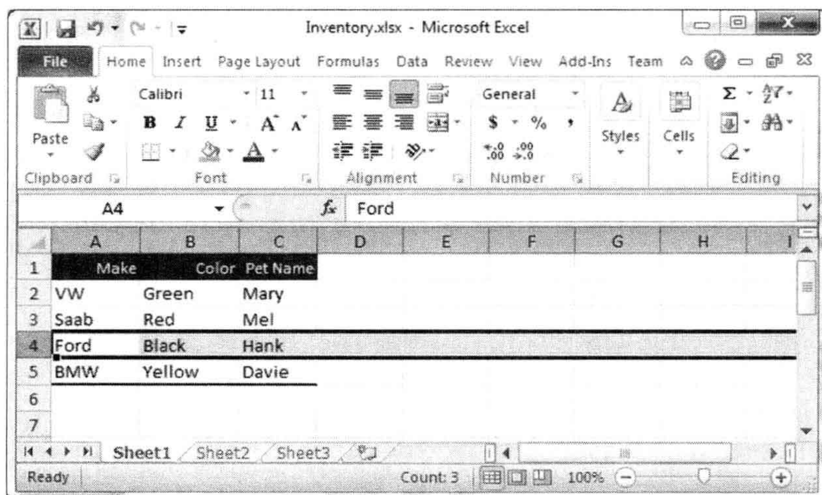


图16-10　将数据导出到Excel文件

16.6　不使用 C# 动态数据进行 COM 互操作

那么，如果你在Solution Explorer中选择Microsoft.Office.Interop.Excel.dll程序集，并将其Embed Interop Type属性设置为False，那么COM Variant将无法识别为动态数据而只能识别为System.Object变量，因此会产生编译时错误。这就需要修改ExportToExcel()方法，添加一些显式转换操作。

如果该项目在.NET 3.5或更早的版本下编译，你将无法使用可选参数和命名参数，而必须显式标记所有缺失的参数。使用早期版本的C#所编写的ExportToExcel()方法如下（注意代码增加的复杂性）：

```
static void ExportToExcel2008(List<Car> carsInStock)
{
  Excel.Application excelApp = new Excel.Application();

  // 必须标记缺失的参数
  excelApp.Workbooks.Add(Type.Missing);

  // 必须将Object转换为_Worksheet
  Excel._Worksheet workSheet = (Excel._Worksheet)excelApp.ActiveSheet;

  // 必须将每个Object转换为Range对象，然后调用低等级的Value2属性
  ((Excel.Range)excelApp.Cells[1, "A"]).Value2 = "Make";
  ((Excel.Range)excelApp.Cells[1, "B"]).Value2 = "Color";
  ((Excel.Range)excelApp.Cells[1, "C"]).Value2 = "Pet Name";

  int row = 1;
  foreach (Car c in carsInStock)
  {
    row++;
    // 必须将每个Object转换为Range，然后调用低等级的Value2属性
    ((Excel.Range)workSheet.Cells[row, "A"]).Value2 = c.Make;
    ((Excel.Range)workSheet.Cells[row, "B"]).Value2 = c.Color;
    ((Excel.Range)workSheet.Cells[row, "C"]).Value2 = c.PetName;
  }

  // 必须调用get_Range方法，并指定缺失的参数
  excelApp.get_Range("A1", Type.Missing).AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2,
    Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing);

  // 必须指定所有缺失的参数
  workSheet.SaveAs(string.Format(@"{0}\Inventory.xlsx", Environment.CurrentDirectory),
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing);

  excelApp.Quit();
  MessageBox.Show("The Inventory.xslx file has been saved to your app folder",
    "Export complete!");
}
```

尽管该程序运行的最终结果相同，但该方法要啰嗦得多，我相信你也一定会这么认为。C# dynamic关键字和DLR的讲解就到此结束了。希望你已经明白这些特性是如何简化复杂的编程任务的。或许更重要的一点是，在使用过程中知道如何折中。选择动态数据难免会丧失类型安全，并且代码也容易出现运行时错误。

尽管关于DLR我有很多想说的，但本章主要关注对日常编程来说更实用和更有帮助的话题。如果要学习DLR的高级特性（如与脚本语言集成），请参考.NET Framework 4.5 SDK文档（从“Dynamic Language Runtime Overview”话题开始）。

源代码 ExportDataToOfficeApp项目的源代码位于Chapter16子目录下。

16.7 小结

C#4.0引入的dynamic关键字可以创建动态数据，在运行时之前我们无法知道这些动态数据的真正身份。当使用动态语言运行时（DLR）进行处理时，自动创建的“表达式树”将被传递给正确的动态语言绑定器。绑定器解析表达式树并传递给正确的对象成员。

使用动态数据和DLR可以从根本上简化很多复杂的C#编程任务，尤其是能够简化在.NET应用程序中使用COM库的过程。正如你在本章中所看到的那样，.NET 4.0和更高版本提供了大量简化COM互操作的特性（与静态数据无关），如在应用程序中嵌入COM互操作数据、可选参数和命名参数。

尽管这些特性可以使我们的代码变得简单明了，但要记住的是动态数据会降低C#代码的类型安全性，并且会带来运行时错误。因此在C#项目中使用动态数据时务必要权衡利弊，并做好相关测试。

第17章

进程、应用程序域和对象上下文

第14章和第15章研究了CLR解析外部引用程序集位置的步骤以及.NET元数据的作用。本章将深入探讨程序集由CLR承载的细节，并阐述进程、应用程序域和对象上下文（context）之间的关系。

简单地说，进程可以承载一组相关的.NET程序集，而应用程序域（简称AppDomain）是对该进程的逻辑细分。你将看到，一个应用程序域进一步被细分成多个*上下文边界*，这些边界用来分组目的相似的.NET对象。使用上下文的概念，CLR便能够确保恰当地控制那些带特殊运行时要求的对象。

尽管大多数的日常编程任务可能不会直接使用进程、应用程序域或对象上下文，但在使用很多.NET API时，理解这些话题是非常重要的，如WCF、多线程和并行处理以及对象序列化。

17.1 Windows 进程的作用

在.NET平台发布之前，进程的概念已经在Windows操作系统中存在很久了。简单地说，进程是一个运行程序。正式地说，进程是一个操作系统级别的概念，用来描述一组资源（比如外部代码库和主线程）和程序运行所必需的内存分配。对于每一个被加载到内存的*.exe，在它的生命周期中操作系统会为之创建一个单独且隔离的进程。

由于一个进程的失败不会影响其他进程，使用这种隔离方式，运行库环境将更加健壮和稳定。此外，一个进程无法访问另外一个进程中的数据，除非使用WCF这种分布式计算编程的API。鉴于以上几点，可以将进程理解为一个正在运行的应用程序的固定的安全的边界。

现在每一个Windows进程都有一个唯一的进程标识符（PID），当需要时，它们能被操作系统加载或卸载（也可通过编程调用）。读者可能知道，在Windows任务管理器（可通过Ctrl+Shift+Esc组合键[①]激活）的Processes选项卡中，我们能够看到机器上正在运行进程的统计信息，这些信息包括进程标识符（PID）和映像名称（image name），如图17-1所示。

说明 默认情况下，不显示Process选项卡中的PID列。通过在Windows任务管理器上选择View→Select Columns菜单项，可以选择想要显示的PID复选框。

---

① 或Ctrl+Alt+Delete组合键。——编者注

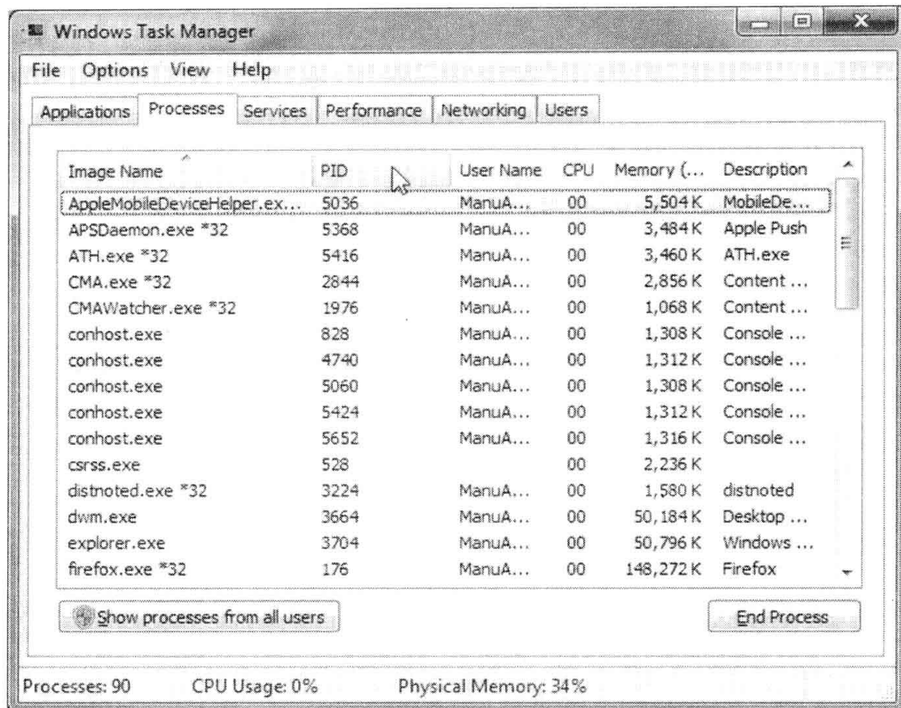


图17-1 Windows任务管理器

线程的作用

每一个Windows进程都恰好包含一个用做程序入口点（entry point）的主线程。在第19章中，我们将研究在.NET平台下构建多线程应用程序的细节。然而，为了使本章的这些主题更容易理解，我们必须先做一些有关线程的定义。首先，线程是进程中的基本执行单元（a path of execution）。正式的说法是：进程的入口点创建的第一个线程被称为*主线程*。.NET执行程序（控制台应用程序、Windows Forms应用程序、WPF应用程序等）使用`Main()`方法作为程序入口点。当调用该方法时，会自动创建全线程。

仅包含一个主线程的进程是*线程安全的*，这是由于在某个特定时刻只有一个线程访问程序中的数据。然而，如果这个线程正在执行一个复杂的操作（比如：输出一个冗长的文本文件，运行一个耗时的计算或尝试连接一个数千公里外的远程服务器），那么这个线程所在的进程（特别是GUI程序）对用户来说会显得像没有响应一样。

由于单线程程序的这个潜在缺陷，Windows API（和.NET平台）可让主线程使用如`CreateThread()`之类的Windows API函数另外产生次线程（术语也称为*工作者线程*，worker thread）。每一个线程（无论主线程还是次线程）都是进程中的一个独立执行单元，它们能够同时访问那些共享数据。

可以猜到，开发者使用多线程有助于改善程序的总体响应性。包含多个线程的进程给我们带来这样一种感觉，即大量的活动几乎在同一时间发生。比如：一个应用程序可以产生一个工作者线程来执

行强度大的工作（比如输出大文本文件）。当这个次线程正在忙碌的时候，主线程仍然对用户的输入保持响应，这使得整个进程具有更强的性能。然而，事情也不尽然，如果单个进程中的线程过多的话，性能反而会下降，因为CPU需要花费不少时间在这些活动的线程之间来回切换。

在一些机器上，多线程是操作系统带给我们最常见的一个假象。单CPU的计算机并没有能力在同一时间运行多个线程。确切地说，在一个单位时间（即一个时间片）内，单CPU只能根据线程优先级执行一个线程。当一个线程的时间片用完的时候，它会被挂起，以便执行其他线程。对于线程来说，它们需要在挂起前记住发生了什么，它们把这些情况写到线程本地存储（Thread Local Storage，TLS）中，并且它们还要获得一个独立的调用栈（call stack），如图17-2所示。

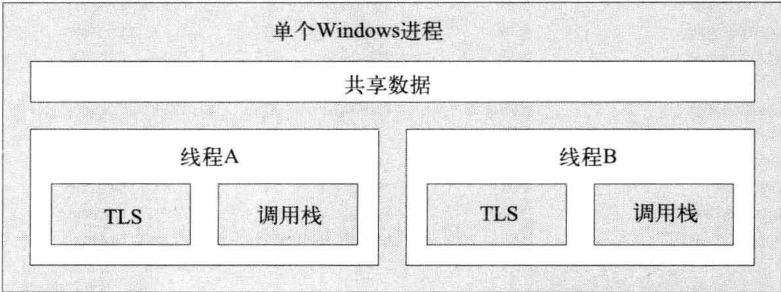


图 17-2　Windows 进程和线程关系

如果线程的概念对于你来说很新，那么不必考虑过多细节。我们只需要记住：线程是Windows进程中的独立执行单元。每一个进程都有一个（在可执行入口点处创建的）主线程，并且每个进程还可以包含以编程方式创建的额外线程。

17.2　.NET 平台下与进程进行交互

虽然进程和线程并不是什么新概念，但是在.NET平台下使用它们的方式改变了很多（也变得更好）。为了后面更好地理解如何构建多线程程序集（参见第19章），让我们先来学习如何使用.NET基础类库来和进程进行交互。

`System.Diagnostics`命名空间定义了许多类型，它们允许我们以编程方式访问进程和许多与诊断（diagnostic）相关的类型，比如系统事务日志和性能计数器。在本章中，我们只和定义在表17-1中的与进程相关的类型打交道。

表17-1　System.Diagnostics命名空间中的部分成员

System.Diagnostics命名空间中进程相关的类型	作　用
Process	提供了访问本地和远程进程的功能，允许通过编程方式开始或结束进程
ProcessModule	代表一个加载到特定进程的模块（*.dll或*.exe）。它能够表示任何模块——基于COM、基于.NET或基于传统C的二进制程序

（续）

System.Diagnostics命名空间中进程相关的类型	作　用
ProcessModuleCollection	提供ProcessModule对象的强类型集合
ProcessStartInfo	指定通过Process.Start()方法启动进程时使用的一组值
ProcessThread	代表指定进程中的线程。它用于诊断一个进程的线程情况，并不用于在进程中创建线程
ProcessThreadCollection	提供ProcessThread对象的强类型集合

System.Diagnostics.Process类用于分析运行于（本地或远程的）机器上的进程。Process类也提供了成员，可用来以编程方式开始、结束进程，设定进程的优先级，以及获得进程中活动线程的列表并且加载给定进程的模块。表17-2显示了System.Diagnostics.Process的部分关键成员。

表17-2 Process类型的部分成员

成　员	作　用
ExitTime	获取终止进程相关的时间戳（类型是DateTime）
Handle	返回操作系统分配给进程的句柄（由IntPtr表示）。当构建与非托管代码交互的.NET程序时，该属性很有用
Id	获取关联进程的PID
MachineName	获取关联进程运行的计算机名称
MainWindowTitle	获取进程主窗口的标题（如果进程没有主窗口，则返回空字符串）
Modules	可以访问强类型ProcessModuleCollection，后者表示一组加载到当前进程的模块（*.dll或*.exe）
ProcessName	获取进程的名称（也就是应用程序本身的名称）
Responding	指示进程的用户界面是否响应用户输入（或者当前是否被"挂起"）
StartTime	获取关联进程开始的时间（通过DateTime类型）
Threads	获取运行在关联进程中的一组线程的设置（由ProcessThread对象的集合表示）

除了上面的属性外，System.Diagnostics.Process还定义了一些有用的方法（见表17-3）。

表17-3 Process类型的部分方法

成　员	作　用
CloseMainWindow()	通过向进程的主窗口发送关闭消息来关闭拥有用户界面的进程
GetCurrentProcess()	这个静态方法返回新的Process对象以表示当前活动的进程
GetProcesses()	这个静态方法返回运行在给定计算机上的新Process对象
Kill()	立即停止关联的进程
Start()	启动一个进程

17.2.1 列举运行中的进程

为了说明如何操作Process类型（请原谅冗长），假定你有一个名为ProcessManipulator的C#控制台程序，它定义了Program类中以下静态辅助方法（请确保引入了System.Diagnostics命名空间）：

```
static void ListAllRunningProcesses()
{
  // 得到本机的所有进程，按PID顺序
  var runningProcs =
    from proc in Process.GetProcesses(".") orderby proc.Id select proc;

  // 输出每个进程的PID和名称
  foreach(var p in runningProcs)
  {
    string info = string.Format("-> PID: {0}\tName: {1}",
      p.Id, p.ProcessName);
    Console.WriteLine(info);
  }
  Console.WriteLine("************************************\n");
}
```

注意，静态方法Process.GetProcesses()返回了一个Process对象数组，这个数组代表目标机器上运行的进程（这里显示的"."符号表示本机）。获得了Process对象数组后，便可以调用表17-2和表17-3中所列的成员了。在本例中，我们只显示每个进程的PID和名称，这些进程按PID排序。如果修改Main()方法调用ListAllRunningProcesses()的话，将看到如下所示的输出：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Processes *****\n");
  ListAllRunningProcesses();
  Console.ReadLine();
}
```

你将在本地机器上看到所有进程的名称和PID。下面是我的机器的部分输出结果：

```
***** Fun with Processes *****

-> PID: 0       Name: Idle
-> PID: 4       Name: System
-> PID: 108     Name: iexplore
-> PID: 268     Name: smss
-> PID: 432     Name: csrss
-> PID: 448     Name: svchost
-> PID: 472     Name: wininit
-> PID: 504     Name: csrss
-> PID: 536     Name: winlogon
-> PID: 560     Name: services
-> PID: 584     Name: lsass
-> PID: 592     Name: lsm
-> PID: 660     Name: devenv
-> PID: 684     Name: svchost
-> PID: 760     Name: svchost
-> PID: 832     Name: svchost
-> PID: 844     Name: svchost
-> PID: 856     Name: svchost
```

```
-> PID: 900      Name: svchost
-> PID: 924      Name: svchost
-> PID: 956      Name: VMwareService
-> PID: 1116     Name: spoolsv
-> PID: 1136     Name: ProcessManipulator.vshost
************************************
```

17.2.2 特定的进程

除了能够获得给定机器上所有运行的进程列表以外，静态方法Process.GetProcessById()还可以通过关联的PID获得单个Process对象。如果请求的PID不存在，就会引发ArgumentException异常。因此，如果想获得进程PID为987的Process对象，可编写如下代码：

```
// 如果PID为987的进程不存在，将会引发运行时错误
static void GetSpecificProcess()
{
  Process theProc =  null;
  try
  {
    theProc = Process.GetProcessById(987);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

至此，你已经学习了如何获取所有进程的列表，或通过PID查找某台机器上的特定进程。尽管了解PID和进程名称也是很有用的，但Process类还能查找给定进程中正在使用的线程和库的集合。让我们来看看如何实现。

17.2.3 进程的线程集合

这组线程通过ProcessThreadCollection强类型集合来表示，包含了许多单个的ProcessThread对象。为说明这一点，假定下面的静态辅助函数已经加到程序中去了：

```
static void EnumThreadsForPid(int pID)
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(pID);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }

  // 列出指定进程中每个线程的统计数字
  Console.WriteLine("Here are the threads used by: {0}",
    theProc.ProcessName);
  ProcessThreadCollection theThreads = theProc.Threads;
```

```
  foreach(ProcessThread pt in theThreads)
  {
    string info =
      string.Format("-> Thread ID: {0}\tStart Time: {1}\tPriority: {2}",
        pt.Id , pt.StartTime.ToShortTimeString(), pt.PriorityLevel);
    Console.WriteLine(info);
  }
  Console.WriteLine("************************************\n");
}
```

可见，System.Diagnostics.Process类型的Threads属性可以访问ProcessThreadCollection类。这里输出了客户指定进程的每个线程分配的线程ID、开始时间、优先级等。现在，如果修改程序中的Main()方法提示用户输入一个PID并进行研究，如下所示：

```
static void Main(string[] args)
{
...
  // 提示用户输入PID，并输出一组活动的线程①
  Console.WriteLine("***** Enter PID of process to investigate *****");
  Console.Write("PID: ");
  string pID = Console.ReadLine();
  int theProcID = int.Parse(pID);

  EnumThreadsForPid(theProcID);
  Console.ReadLine();
}
```

运行程序，输入机器中任意进程的PID，就可以查看其线程了。下面的输出结果显示了我机器上PID为108的线程，它承载了微软Internet浏览器：

```
***** Enter PID of process to investigate *****
PID: 108
Here are the threads used by: iexplore
-> Thread ID: 680     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2040    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 880     Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3380    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3376    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3448    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3476    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2264    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2380    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2384    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 2308    Start Time: 9:05 AM     Priority: Normal
-> Thread ID: 3096    Start Time: 9:07 AM     Priority: Highest
-> Thread ID: 3600    Start Time: 9:45 AM     Priority: Normal
-> Thread ID: 1412    Start Time: 10:02 AM    Priority: Normal
```

除了Id、StartTime和PriorityLevel外，ProcessThread类型还有一些有趣的成员。表17-4列出了一些值得关注的成员。

① 请勿输入PID为0的System Idle Process的“进程”，否则会出错。因为它并不是一个真正意义上的进程，而是系统核心虚拟出来的表示CPU空闲的状态。——译者注

表17-4 ProcessThread类型的部分成员

成　　员	作　　用
CurrentPriority	获取线程的当前优先级
Id	获取线程的唯一标识符
IdealProcessor	设置线程运行的首选处理器
PriorityLevel	获取或设置线程的优先级别
ProcessorAffinity	设置关联线程可以运行的处理器
StartAddress	获取操作系统启动线程要调用的函数的内存地址
StartTime	获取操作系统启动线程的时间
ThreadState	获取线程的当前状态
TotalProcessorTime	获取线程使用处理器的时间总量
WaitReason	获取线程等待的原因

在继续阅读之前，请记住在.NET平台下，ProcessThread类型并不用于创建、挂起或停止线程，ProcessThread是一种用来获取运行进程中活动Windows线程诊断信息的手段。在第19章中，我们将研究如何通过System.Threading命名空间来构建多线程应用程序。

17.2.4 进程中的模块集合

接下来，看看如何对承载在进程中的加载模块的数量进行迭代。回想一下，模块这个词用于描绘承载于指定进程中的*.dll（或*.exe本身）。当通过Process.Modules属性访问ProcessModuleCollection时，可以列举出承载在进程中的所有模块：基于.NET、基于COM、基于传统C的库。请思考如下的辅助函数，它将通过PID列举出在指定进程中的模块：

```
static void EnumModsForPid(int pID)
{
  Process theProc = null;
  try
  {
    theProc = Process.GetProcessById(pID);
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }

  Console.WriteLine("Here are the loaded modules for: {0}",
    theProc.ProcessName);
  ProcessModuleCollection theMods = theProc.Modules;
  foreach(ProcessModule pm in theMods)
  {
    string info = string.Format("-> Mod Name: {0}", pm.ModuleName);
    Console.WriteLine(info);
  }
  Console.WriteLine("************************************\n");
}
```

为了看到输出结果，请检查承载当前控制台程序（ProcessManipulator）的进程中加载的模块。为此，我们运行程序，通过任务管理器找到赋给ProcessManipulator.exe的PID值，将值传入EnumModsForPid()方法（请确保Main()方法得到了同步更新）。最后，会惊讶地看到这个简单控制台程序所使用的*.dll列表（GDI32.dll、USER32.dll、ole32.dll等）。输出结果如下所示：

```
Here are the loaded modules for: ProcessManipulator
-> Mod Name: ProcessManipulator.exe
-> Mod Name: ntdll.dll
-> Mod Name: MSCOREE.DLL
-> Mod Name: KERNEL32.dll
-> Mod Name: KERNELBASE.dll
-> Mod Name: ADVAPI32.dll
-> Mod Name: msvcrt.dll
-> Mod Name: sechost.dll
-> Mod Name: RPCRT4.dll
-> Mod Name: SspiCli.dll
-> Mod Name: CRYPTBASE.dll
-> Mod Name: mscoreei.dll
-> Mod Name: SHLWAPI.dll
-> Mod Name: GDI32.dll
-> Mod Name: USER32.dll
-> Mod Name: LPK.dll
-> Mod Name: USP10.dll
-> Mod Name: IMM32.DLL
-> Mod Name: MSCTF.dll
-> Mod Name: clr.dll
-> Mod Name: MSVCR100_CLR0400.dll
-> Mod Name: mscorlib.ni.dll
-> Mod Name: nlssorting.dll
-> Mod Name: ole32.dll
-> Mod Name: clrjit.dll
-> Mod Name: System.ni.dll
-> Mod Name: System.Core.ni.dll
-> Mod Name: psapi.dll
-> Mod Name: shfolder.dll
-> Mod Name: SHELL32.dll
************************************
```

17.2.5　以编程方式启动或结束进程

对于System.Diagnostics.Process类，最后再来看看它的Start()和Kill()方法。仅从名字上便能看出，它们提供了启动和结束进程的编程方法。例如，考虑下面的静态辅助方法StartAndKillProcess()：

```
static void StartAndKillProcess()
{
  Process ieProc = null;

  // 启动IE，进入Facebook
  try
  {
    ieProc = Process.Start("IExplore.exe", "www.facebook.com");
```

```
    }
    catch (InvalidOperationException ex)
    {
      Console.WriteLine(ex.Message);
    }

    Console.Write("--> Hit enter to kill {0}...", ieProc.ProcessName);
    Console.ReadLine();

    // 结束iexplorer.exe进程
    try
    {
      ieProc.Kill();
    }
    catch (InvalidOperationException ex)
    {
      Console.WriteLine(ex.Message);
    }
  }
```

说明 为了启动新的进程，必须以管理员权限运行Visual Studio，否则将得到运行时错误。

静态方法Process.Start()有几次重载。至少需要指定希望加载的进程的友好名称（比如：Microsoft Internet Explorer、iexplore.exe）。本例中，我们使用了Start()方法的变体，该方法允许我们向程序的入口点（即Main()方法）传入任意额外的参数。

调用Start()方法后，将返回刚刚激活的进程的引用。要终止程序，可以调用实例级别的Kill()方法。在这里，我们用try/catch块包裹Start()和Kill()的调用，并处理InvalidOperationException错误。这对Kill()方法的调用来说尤为重要，因为如果进程在调用Kill()之前就已经被终止，则会触发该错误。

17.2.6 使用ProcessStartInfo类控制进程的启动

Start()方法还允许传入一个System.Diagnostics.ProcessStartInfo类型，说明一些关于进程如何被激活的额外信息。下面是ProcessStartInfo的正式定义（完整的细节请看.NET Framework 4.5 SDK文档）。

```
public sealed class ProcessStartInfo : object
{
  public ProcessStartInfo();
  public ProcessStartInfo(string fileName);
  public ProcessStartInfo(string fileName, string arguments);
  public string Arguments { get; set; }
  public bool CreateNoWindow { get; set; }
  public StringDictionary EnvironmentVariables { get; }
  public bool ErrorDialog { get; set; }
  public IntPtr ErrorDialogParentHandle { get; set; }
  public string FileName { get; set; }
  public bool LoadUserProfile { get; set; }
  public SecureString Password { get; set; }
  public bool RedirectStandardError { get; set; }
  public bool RedirectStandardInput { get; set; }
  public bool RedirectStandardOutput { get; set; }
  public Encoding StandardErrorEncoding { get; set; }
```

```
  public Encoding StandardOutputEncoding { get; set; }
  public bool UseShellExecute { get; set; }
  public string Verb { get; set; }
  public string[] Verbs { get; }
  public ProcessWindowStyle WindowStyle { get; set; }
  public string WorkingDirectory { get; set; }
}
```

为了演示如何控制进程的启动，我们修改了StartAndKillProcess()，让它加载微软Internet浏览器，导航到www.facebook.com，并且显示窗口的最大化状态：

```
tatic void StartAndKillProcess()
{
  Process ieProc = null;

  // 用最大化窗口启动IE，并导航到facebook
  try
  {
    ProcessStartInfo startInfo = new
      ProcessStartInfo("IExplore.exe", "www.facebook.com");
    startInfo.WindowStyle = ProcessWindowStyle.Maximized;

    ieProc = Process.Start(startInfo);
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine(ex.Message);
  }
...
}
```

现在你已经理解了Windows进程的作用以及如何在C#代码中与之交互，你已经为研究.NET应用程序域的概念作好了准备。

源代码　ProcessManipulator项目的源代码位于Chapter 17子目录下。

17.3　.NET 应用程序域

在.NET平台下，可执行程序并没有直接承载在Windows进程中，而传统的非托管程序是直接承载的。实际上，.NET可执行程序承载在进程的一个逻辑分区中，术语称为*应用程序域*（AppDomain）。可见，一个进程可以包含多个应用程序域，每一个应用程序域中承载一个.NET可执行程序。这种对传统的Windows进程的进一步分区具有几个好处，下面列出一些。

- 应用程序域是.NET平台操作系统独立性的关键特性。这种逻辑分区将不同操作系统表现加载可执行程序的差异抽象化了。
- 和一个完整的进程相比，应用程序域的CPU和内存占用都要小得多。因此CLR加载和卸载应用程序域比起完整的进程来说也快得多，并且可以快速提升服务器应用程序的可扩展性。
- 应用程序域为承载的应用程序提供了深度的隔离。如果进程中一个应用程序域失败了，剩余的应用程序域也能保持正常。

前面已经提到，单个进程可以承载多个应用程序域，其中每一个程序域都和该进程（或其他进程）中其他的程序域完全彻底隔离开。由此，如果不使用分布式编程协议（如WCF），运行在某个应用程序域中的应用程序将无法访问其他应用程序域中的任何数据（无论是全局变量还是静态字段）。

虽然单个进程可以承载多个应用程序域，但是情况也有例外。至少操作系统进程只能承载默认的应用程序域。在进程启动的时候，CLR将自动创建这个特定的应用程序域。此后，CLR能够根据需求创建其他的应用程序域。

System.AppDomain类

.NET平台允许我们使用mscorlib.dll中System命名空间下的AppDomain类，以编程的方式监控应用程序域、在运行时新建应用程序域、向应用程序域加载程序集等多种任务。表17-5列出了AppDomain类中的一些方法（详细内容可参考.NET Framework 4.5 SDK文档）。

表17-5 AppDomain的主要成员

成　员	作　用
CreateDomain()	该静态方法在当前进程中创建一个新的应用程序域
CreateInstance()	在加载程序集到调用的应用程序域时，在外部程序集文件中创建指定类型的新实例
ExecuteAssembly()	根据文件名在应用程序域中执行程序集
GetAssemblies()	获取已经加载到此应用程序域中的.NET程序集（基于COM和C的二进制文件除外）
GetCurrentThreadId()	该静态方法返回当前应用程序域上活动的线程ID
Load()	动态加载程序集到当前的应用程序域
Unload()	该静态方法在进程中卸载指定的应用程序域

说明 .NET平台不允许从内存中卸载指定的程序集。以编程方式卸载库的唯一方式是使用Unload()方法销毁承载的应用程序域。

此外，AppDomain类还定义了一些属性，用来监控给定应用程序域的活动。表17-6列出了一些核心属性。

表17-6 AppDomain的部分属性

属　性	作　用
BaseDirectory	获取目录路径，程序集解决程序用它来探测程序集
CurrentDomain	该静态属性获取当前执行线程所在的应用程序域
FriendlyName	获取当前应用程序域的友好名称
MonitoringIsEnabled	获取或设置一个值，该值指示是否对当前进程启用应用程序域的CPU和内存监控。一旦对进程启用了监控，则无法将其禁用
SetupInformation	获取给定应用程序域的配置信息，表示为一个AppDomainSetup对象

最后，AppDomain类支持一组事件，对应于应用程序域生命周期中的不同部分。表17-7显示了你很可能会用到的事件。

表17-7　AppDomain类型的部分事件

事　　件	作　　用
AssemblyLoad	在加载程序集到内存时发生
AssemblyResolve	在对程序集的解析失败时发生
DomainUnload	在即将从主进程中卸载AppDomain时发生
FirstChanceException	在应用程序域抛出异常时，该事件将在CLR找到合适的catch语句之前触发
ProcessExit	当默认应用程序域的父进程退出时，在默认应用程序域上发生
UnhandledException	在异常处理程序未捕捉到异常时发生

17.4　与默认应用程序域进行交互

当一个.NET可执行文件启动时，CLR会自动将其放置到宿主进程的默认应用程序域中。该过程是自动且透明的，你不需要编写任何代码。但你可以使用AppDomain.CurrentDomain属性来访问这个默认的应用程序域。有了这个访问点，就可以捕获任何感兴趣的事件，或使用AppDomain中的方法和属性来执行运行时诊断。

为了学习如何与默认应用程序域交互，我们新建一个控制台应用程序，取名为DefaultAppDomain-App。使用以下逻辑更新Program类，它将使用AppDomain类中的一些成员显示默认应用程序域中的某些细节：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with the default AppDomain *****\n");
    DisplayDADStats();
    Console.ReadLine();
  }

  private static void DisplayDADStats()
  {
    // 访问当前线程的应用程序域
    AppDomain defaultAD = AppDomain.CurrentDomain;

    // 打印该域中的不同状态
    Console.WriteLine("Name of this domain: {0}", defaultAD.FriendlyName);
    Console.WriteLine("ID of domain in this process: {0}", defaultAD.Id);
    Console.WriteLine("Is this the default domain?: {0}",
      defaultAD.IsDefaultAppDomain());
    Console.WriteLine("Base directory of this domain: {0}", defaultAD.BaseDirectory);
  }
}
```

该示例的输出结果如下：

```
***** Fun with the default AppDomain *****

Name of this domain: DefaultAppDomainApp.exe
ID of domain in this process: 1
Is this the default domain?: True
Base directory of this domain: E:\MyCode\DefaultAppDomainApp\bin\Debug\
```

注意，默认应用程序域的名称与包含在其内的可执行文件的名称相同（本例为DefaultApp-DomainApp.exe）。同时还要注意，用来探测所需要的外部私有程序集的基目录值，映射为可执行文件被部署的位置。

17.4.1 枚举加载的程序集

你还可以使用GetAssemblies()实例方法，获取给定应用程序域中所加载的.NET程序集。该方法返回Assembly对象的数组，这在第15章已经介绍过了，该数组是System.Reflection命名空间的成员（所以要记得在C#代码文件中引入该命名空间）。

我们在Program类中定义了一个新方法ListAllAssembliesInAppDomain()。该辅助方法获取所有加载的程序集，并打印它们的友好名称和版本号：

```
static void ListAllAssembliesInAppDomain()
{
  // 访问当前线程的应用程序域
  AppDomain defaultAD = AppDomain.CurrentDomain;

  // 获取默认应用程序域中所有加载的程序集
  Assembly[] loadedAssemblies = defaultAD.GetAssemblies();
  Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
    defaultAD.FriendlyName);
  foreach(Assembly a in loadedAssemblies)
  {
    Console.WriteLine("-> Name: {0}", a.GetName().Name);
    Console.WriteLine("-> Version: {0}\n", a.GetName().Version);
  }
}
```

更新Main()方法以调用该成员，可以看到承载可执行文件的应用程序域使用了如下所示的.NET程序集：

```
***** Here are the assemblies loaded in DefaultAppDomainApp.exe *****

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: DefaultAppDomainApp
-> Version: 1.0.0.0
```

所加载的程序集的列表可因编写新的C#代码而随时改变。例如，假设我们更新ListAllAssem-bliesInAppDomain()方法，使其使用LINQ查询对加载的程序集按名称排序：

```
static void ListAllAssembliesInAppDomain()
{
  // 访问当前线程的应用程序域
  AppDomain defaultAD = AppDomain.CurrentDomain;

  // 获取默认应用程序域中所有加载的程序集
  var loadedAssemblies = from a in defaultAD.GetAssemblies()
    orderby a.GetName().Name select a;

  Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
      defaultAD.FriendlyName);
  foreach (var a in loadedAssemblies)
  {
    Console.WriteLine("-> Name: {0}", a.GetName().Name);
    Console.WriteLine("-> Version: {0}\n", a.GetName().Version);
  }
}
```

再次运行该程序，可以发现System.Core.dll和System.dll也加载到了内存中，因为它们是LINQ to Object API所必需的：

```
***** Here are the assemblies loaded in DefaultAppDomainApp.exe *****

-> Name: DefaultAppDomainApp
-> Version: 1.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: System
-> Version: 4.0.0.0

-> Name: System.Core
-> Version: 4.0.0.0
```

17.4.2 接收程序集加载通知

如果想接收CLR在向给定的应用程序域中加载新程序集时所发出的通知，可以处理AssemblyLoad事件。该事件的类型为AssemblyLoadEventHandler委托，它所指向方法的第一个参数为System.Object类型，第二个参数为AssemblyLoadEventArgs类型。

让我们再为当前的Program类添加最后一个方法InitDAD()。顾名思义，该方法将初始化默认的应用程序域，用一个Lambda表达式处理AssemblyLoad事件：

```
private static void InitDAD()
{
  // 这段逻辑将在应用程序域创建后，打印加载到应用程序域的程序集名称
  AppDomain defaultAD = AppDomain.CurrentDomain;
  defaultAD.AssemblyLoad += (o, s) =>
    {
      Console.WriteLine("{0} has been loaded!", s.LoadedAssembly.GetName().Name);
    };
}
```

运行修改后的应用程序，加载新的程序集时将通知我们。这里我们使用传入的AssemblyLoaded-EventArgs参数的LoadedAssembly属性简单地打印了程序集的友好名称。

源代码 DefaultAppDomainApp项目的源代码位于Chapter 17子目录下。

17.5 创建新的应用程序域

单个进程可以通过AppDomain.CreateDomain()静态方法承载多个应用程序域。尽管对大多数.NET应用程序来说，在运行时新建应用程序域是十分罕见的，但理解其原理还是非常重要的。例如，在本章稍后你将看到，构建动态程序集（见第18章）时，需要将它们安装到自定义的应用程序域中。同样，很多.NET安全API都要求了解如何构建新的应用程序域，以便基于提供的安全证书来隔离程序集。

为了研究如何在运行时新建应用程序域（以及如何在这些自定义宿主中加载程序集），我们新建一个控制台应用程序，取名为CustomAppDomains。AppDomain.CreateDomain()方法包含多个重载。你至少需要指定新应用程序域的友好名称。使用如下代码更新Program类：这里我们利用前面示例中的ListAllAssembliesInAppDomain()方法，但这次我们将AppDomain对象作为传入参数进行分析：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Custom AppDomains *****\n");

    // 显示默认应用程序域中加载的所有程序集
    AppDomain defaultAD = AppDomain.CurrentDomain;
    ListAllAssembliesInAppDomain(defaultAD);

    // 创建一个新的应用程序域
    MakeNewAppDomain();
    Console.ReadLine();
  }

  private static void MakeNewAppDomain()
  {
    // 在当前进程中新建一个AppDomain，并列出它所加载的程序集
    AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
    ListAllAssembliesInAppDomain(newAD);
  }

  static void ListAllAssembliesInAppDomain(AppDomain ad)
  {
    // 现在获取默认应用程序域中加载的所有程序集
    var loadedAssemblies = from a in ad.GetAssemblies()
                           orderby a.GetName().Name select a;

    Console.WriteLine("***** Here are the assemblies loaded in {0} *****\n",
      ad.FriendlyName);
    foreach (var a in loadedAssemblies)
    {
```

```
            Console.WriteLine("-> Name: {0}", a.GetName().Name);
            Console.WriteLine("-> Version: {0}\n", a.GetName().Version);
        }
    }
}
```

运行当前示例，你会发现默认应用程序域(CustomAppDomains.exe)加载了mscorlib.dll、System.dll、System.Core.dll和当前项目的代码库CustomAppDomains.exe。而新的应用程序域则只包含mscorlib.dll，CLR总是为每个应用程序域都加载这个.NET程序集：

```
***** Fun with Custom AppDomains *****

***** Here are the assemblies loaded in CustomAppDomains.exe *****

-> Name: CustomAppDomains
-> Version: 1.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: System
-> Version: 4.0.0.0

-> Name: System.Core
-> Version: 4.0.0.0

***** Here are the assemblies loaded in SecondAppDomain *****

-> Name: mscorlib
-> Version: 4.0.0.0
```

说明　调式（按F5键）该程序，你会发现每个应用程序域都加载了很多用于Visual Studio调试进程的额外的程序集。运行该项目（按Ctrl + F5组合键），将只显示每个应用程序域的程序集。

如果你有传统的Windows编程经验，可能会觉得这有悖常理（如你所想的那样，两个应用程序域访问了相同的程序集）。但是，程序集是加载到应用程序域中的，而不是直接加载到进程本身的。

17.5.1　在自定义应用程序域中加载程序集

CLR可随时向默认的应用程序域加载程序集。如果手工创建了应用程序域，可以使用AppDomain.Load()方法向其加载程序集。同样，调用AppDomain.ExecuteAssembly()方法也可以加载一个*.exe程序集，并执行其Main()方法。

假设你要向新建的次应用程序域中加载CarLibrary.dll，并已经将其复制到了当前应用程序的\bin\Debug文件夹下。更新MakeNewAppDomain()方法（确保引入System.IO命名空间以便可以访问FileNotFoundException类）：

```
private static void MakeNewAppDomain()
{
```

```
  // 在当前进程中新建AppDomain
  AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");

  try
  {
    // 将CarLibrary.dll加载到新域中
    newAD.Load("CarLibrary");
  }
  catch (FileNotFoundException ex)
  {
    Console.WriteLine(ex.Message);
  }

  // 列出所有的程序集
  ListAllAssembliesInAppDomain(newAD);
}
```

这时程序的输出结果如下（注意CarLibrary.dll）：

```
***** Fun with Custom AppDomains *****

***** Here are the assemblies loaded in CustomAppDomains.exe *****

-> Name: CustomAppDomains
-> Version: 1.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0

-> Name: System
-> Version: 4.0.0.0

-> Name: System.Core
-> Version: 4.0.0.0

***** Here are the assemblies loaded in SecondAppDomain *****

-> Name: CarLibrary
-> Version: 2.0.0.0

-> Name: mscorlib
-> Version: 4.0.0.0
```

说明 记住，如果调试该应用程序，将看到每个应用程序域中都加载了很多额外的库。

17.5.2 以编程方式卸载应用程序域

需要重点指出的是，CLR并不允许卸载单独的.NET程序集。然而，使用AppDomain.Unload()方法，我们可以选择从承载的进程中卸载指定的应用程序域。当卸载应用程序域时，其中的程序集会依次被卸载。

回想一下，AppDomain类型定义了DomainUnload事件。当自定义应用程序域从包含它的进程中卸载

的时候，该事件被触发。另一个事件是ProcessExit，当默认的应用程序域从进程中卸载的时候（很明显，这是进程本身结束时必须做的事情），它将被触发。

如果想要以编程方式从主进程中卸载newAD，并且在相关应用程序域卸载时得到通知，可以编写如下的事件逻辑来更新MakeNewAppDomain()：

```
private static void MakeNewAppDomain()
{
  // 在当前进程中创建一个新应用程序域
  AppDomain newAD = AppDomain.CreateDomain("SecondAppDomain");
  newAD.DomainUnload += (o, s) =>
  {
    Console.WriteLine("The second AppDomain has been unloaded!");
  };

  try
  {
    // 现在加载CarLibrary.dll到这个新域中
    newAD.Load("CarLibrary");
  }
  catch (FileNotFoundException ex)
  {
    Console.WriteLine(ex.Message);
  }

  // 一一列出所有程序集
  ListAllAssembliesInAppDomain(newAD);

  // 现在卸载这个应用程序域
  AppDomain.Unload(newAD);
}
```

如果想要在默认的应用程序域被卸载时得到通知，请修改Main()方法处理默认应用程序域的ProcessExit事件：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom AppDomains *****\n");

  // 在默认的应用程序域中显示所有加载的程序集
  AppDomain defaultAD = AppDomain.CurrentDomain;
  defaultAD.ProcessExit += (o, s) =>
  {
    Console.WriteLine("Default AD unloaded!");
  };

  ListAllAssembliesInAppDomain(defaultAD);

  MakeNewAppDomain();
  Console.ReadLine();
}
```

我们完成了对.NET应用程序域的介绍。在本章最后，让我们看看一个高级话题，如何将对象按上下文边界进行分组。

源代码　CustomAppDomains项目的源代码位于Chapter 17子目录下。

17.6 对象上下文边界

同刚才看到的一样，应用程序域是承载.NET程序集的进程中的逻辑分区。与此相似，应用程序域也可以进一步被划分成多个上下文边界。简言之，.NET上下文为单独的应用程序域提供了一种方式，该方式能为一个给定对象建立“特定的家”（specific home）。

说明 尽管进程和应用程序域十分重要，但大多数.NET应用程序都不会要求你使用对象上下文。我已经使用这些材料绘制了一幅完整的图画。

使用上下文，CLR可以确保在运行时有特殊需求的对象，可以通过拦截进出上下文的方法调用，得到适当的和一致的处理。这个拦截层允许CLR调整当前的方法调用，以便满足给定对象对上下文的设定要求。举例来说，如果定义一个需要自动线程安全（使用`[Synchronization]`特性）的C#类类型，CLR将会在分配期间创建“上下文同步”。

和一个进程定义了默认的应用程序域一样，每个应用程序域都有一个默认的上下文。这个默认的上下文[由于它总是应用程序域创建的第一个上下文，所以有时称为上下文0（context 0）]用于组合那些对上下文没有具体的或唯一性需求的.NET对象。如你所料，大多数.NET对象都会被加载到上下文0中。如果CLR判断一个新创建的对象有特殊需求，一个新的上下文边界将会在承载它的应用程序域中被创建。图17-3展示了进程/应用程序域/上下文边界之间的关系。

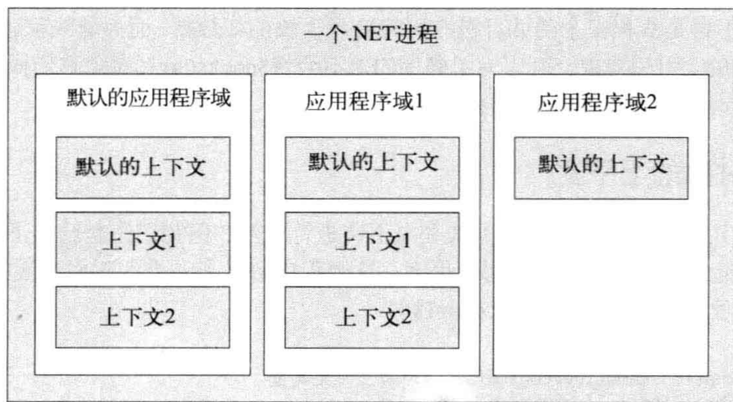


图17-3 进程、应用程序域和上下文边界

17.6.1 上下文灵活和上下文绑定类型

不需要指定特定上下文的.NET类型称为上下文灵活（context-agile）对象。这些对象可以从承载它的应用程序域的任何位置访问，与对象的运行时需求没有关系。想要构建这样一个上下文灵活的对象根本不用费神，因为简单得你什么都不用做（具体来说，不需要修饰类型的上下文特性，也不需要派生自`System.ContextBoundObject`基类）：

```
// 一个上下文灵活的对象被加载到上下文0中
class SportsCar{}
```

另一方面,那些需要上下文分配的对象称为上下文绑定(context-bound)对象,它们必须派生自System.ContextBoundObject基类。这个基类再次说明这样一个事实：任何一个对象都只能在其被创建的那个上下文中正常运行。考虑到.NET上下文的作用，如果一个上下文绑定对象不知何故在一个并不兼容的上下文中终止，则在这种最不合时宜的情况下理应出现错误。

除了派生自System.ContextBoundObject外，一个上下文敏感的类型也可以用特定种类的.NET特性修饰（不要惊讶），术语称为上下文特性。所有的上下文特性派生自ContextAttribute基类。下面来看一个例子。

17.6.2　定义上下文绑定对象

假定想要定义一个自动线程安全的类（SportsCarTS），但又不在成员实现中采用硬编码做线程同步的逻辑，可以继承ContextBoundObject，并应用[Synchronization]特性，如下所示：

```
using System.Runtime.Remoting.Contexts;

// 上下文绑定类型仅仅加载到一个同步的（因此是线程安全的）上下文中
[Synchronization]
class SportsCarTS : ContextBoundObject
{}
```

添加了[Synchronization]特性的类型将被加载到线程安全上下文中。因为SportsCarTs类类型有特殊的需求，如果一个已分配的对象从一个同步的上下文移动到一个非同步的上下文时，发生的问题可想而知。对象突然不再是线程安全的并且极有可能变成大块的坏数据，而大量线程还在试图与这个(现在已是线程不稳定的)引用对象交互。为了确保CLR不会将SportsCarTS对象移出同步上下文边界，只需让SportsCarTS继承自ContextBoundObject。

17.6.3　研究对象的上下文

尽管很少有应用程序需要以编程的方式和上下文进行交互，但我们还是给出下面的示例。该例子创建了一个名为ObjectContextApp的控制台程序，这个程序定义了一个上下文灵活的类（SportsCar）和一个单一的上下文绑定的类型（SportsCarTS）：

```
using System;
using System.Runtime.Remoting.Contexts;  // 为上下文类型
using System.Threading;  // 为线程类型

// SportsCar没有特别的上下文要求，所以将加载到应用程序域的默认上下文中
class SportsCar
{
  public SportsCar()
  {
    // 得到上下文信息，并输出上下文ID
    Context ctx = Thread.CurrentContext;
    Console.WriteLine("{0} object in context {1}",
      this.ToString(), ctx.ContextID);
    foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
      Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
```

```
    }
  }

  // SportsCarTS需要加载到一个同步上下文中
   [Synchronization]
  class SportsCarTS : ContextBoundObject
  {
    public SportsCarTS()
    {
      // 得到上下文信息，并输出上下文ID
      Context ctx = Thread.CurrentContext;
      Console.WriteLine("{0} object in context {1}",
        this.ToString(), ctx.ContextID);
      foreach(IContextProperty itfCtxProp in ctx.ContextProperties)
        Console.WriteLine("-> Ctx Prop: {0}", itfCtxProp.Name);
    }
  }
```

注意，通过调用静态的Thread.CurrentContext属性，SportsCar的每个构造函数都从当前运行的线程中获得了Context对象。使用这个Context对象，我们能够输出上下文边界的统计信息，比如它分配的ID以及一组通过Context.ContextProperties得到的描述。该属性返回一个实现了IContextProperty接口的对象数组，该对象数组通过Name属性公开每一个描述符。现在更新Main()方法，分派每个类类型的实例：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Context *****\n");

  // 对象将显示创建时的上下文信息
  SportsCar sport = new SportsCar();
  Console.WriteLine();

  SportsCar sport2 = new SportsCar();
  Console.WriteLine();

  SportsCarTS synchroSport = new SportsCarTS();
  Console.ReadLine();
}
```

当对象被实例化的时候，类的构造函数将会输出各种上下文信息（输出的“lease life time service property”是.NET remoting层的底层部分，可以忽略）。

```
***** Fun with Object Context *****

ObjectContextApp.SportsCar object in context 0
-> Ctx Prop: LeaseLifeTimeServiceProperty

ObjectContextApp.SportsCar object in context 0
-> Ctx Prop: LeaseLifeTimeServiceProperty

ObjectContextApp.SportsCarTS object in context 1
-> Ctx Prop: LeaseLifeTimeServiceProperty
-> Ctx Prop: Synchronization
```

由于SportsCar类没有使用上下文特性修饰，所以CLR将把sport和sport2分配到上下文0（也就是

默认的上下文中）。然而，SportsCarTS对象被加载到唯一的上下文边界（被分配的上下文ID是1），因为实际上这个上下文绑定类型应用了[Synchronization]特性加以修饰。

源代码　Object ContextApp项目的源代码位于Chapter 17子目录下。

17.7　进程、应用程序域和上下文小结

到目前为止，对于.NET程序集如何由CLR承载，读者已经有了很好的认识，总结起来有以下几个要点。

- 一个.NET进程可以承载多个应用程序域。每一个应用程序域可以承载多个相关的.NET程序集，并且可由CLR（或使用System.AppDomain类型通过编程）独立地加载或卸载应用程序域。
- 一个给定的应用程序域中包含一个或多个上下文。使用上下文，CLR能够将“有特殊需求的”对象放置到一个逻辑容器中，确保该对象的运行时需求能够被满足。

如果前面这些内容对读者的水平来讲有点过于低了，不用担心。大多数情况下，CLR可以自动处理进程、应用程序域和上下文的细节。然而，学习这些的好处是这些内容为理解.NET平台下的多线程编程打下了一个坚实的基础。

17.8　小结

本章的目的在于研究.NET平台如何承载.NET可执行映像。可见，长久以来存在的Windows进程概念，在CLR的需求下被改头换面了。一个单进程（能够通过System.Diagnostics.Process类型编程控制）由多个应用程序域组成。应用程序域代表了进程中被隔离的独立的边界。

如你所看到的，一个进程可以承载多个应用程序域，每一个应用程序域都能承载或执行多个相关程序集。此外，一个应用程序域可以包含多个上下文边界。使用这种进一步的类型隔离，CLR可以确保有特定要求的对象被正确处理。

第 18 章

CIL和动态程序集的作用

在构建大型.NET应用程序时，由于C#（或其他托管语言，如Visual Basic）固有的生产效率和易用性，你很可能选择它们进行开发。但我们在第1章中介绍了，托管编译器的作用是将*.cs代码文件翻译为CIL代码、类型元数据和程序集清单。事实上，CIL是一个成熟的.NET编程语言，包含自己的语法、语义和编译器（ilasm.exe）。

本章将介绍这个.NET平台的母语。你将理解CIL指令、CIL特性和CIL操作码之间的区别，还将学习各种CIL编程工具以及.NET程序集正反向工程的作用，然后介绍了使用CIL语法定义命名空间、类型和成员的基础知识，最后研究了System.Reflection.Emit命名空间的作用以及如何在运行时动态构造程序集（使用CIL指令）。

当然，很少有程序员会在日常工作中使用原始的CIL代码。因此，我将在本章开头介绍为什么了解这个底层.NET语言的语法和语义是很重要的。

18.1 学习 CIL 语法的原因

CIL本质上其实就是.NET平台的母语。当开发人员选择一种托管的编程语言（C#、VB、F#、COBOL.NET等）构建.NET程序集时，同这个语言相关联的编译器就会把源代码翻译成CIL。正如其他任何一种编程语言一样，CIL提供了非常多的结构和实现标记。如果考虑到CIL其实也是一种.NET编程语言，那么通过直接使用.NET Framework 4.5 SDK提供的CIL和CIL编译器（ilasm.exe）来开发和构建.NET程序集也就不会让人吃惊了。

事实上很少有人会选择完全使用CIL来构建一个.NET应用程序，不过CIL本身还是非常有趣的，同时也可以把它看成体现程序员智慧的技术。简单地说，对CIL的语法理解的越多，那么就越有可能进入.NET的高级领域。具体一点儿说，理解并掌握CIL的人应该能够达到以下几点。

- 清楚地理解不同的.NET编程语言是如何映射它们各自的关键字到CIL标记上的。
- 反汇编一个已经存在的.NET程序集，直接编辑CIL代码，最后重新编译更改后的代码到.NET二进制文件。例如，有些情况下为了与一些高级COM特性交互，需要修改COM。
- 使用System.Reflection.Emit命名空间构建动态程序集。这个API允许我们在内存中生成.NET程序集，并且可以选择将其持久化到磁盘中。
- 理解那些存在于CIL层中，但是不被高级托管语言所支持的CTS的特性。要知道，CIL是唯一一种允许你访问和使用所有CTS特性的.NET语言。例如，使用纯粹的CIL，就可以定义全局的成员和字段，而这在C#中是不允许的。

最后，再次强调一下，即使你不想在CIL细节上纠缠，也完全有能力掌握C#和.NET的基础类库。从很多方面来看，CIL对于.NET开发人员就好像汇编语言对于一个C++程序员一样。那些精通底层细节的开发人员就可以具备应用高级技术手段来解决手上问题的能力，同时对于编程（运行）环境底层也会有一个深入的理解。如果你愿意接受挑战，那么就让我们一同来揭开CIL的神秘面纱吧。

说明　本章并不是要全面讲述CIL语法和语义的各个方面。如果有这方面的需要，推荐你从ECMA官方网站（www.ecma-international.org）下载官方ECMA标准（ecma-335.pdf）。

18.2　CIL 指令、特性和操作码

当研究CIL这样的底层开发语言时，一定会发现一些非常熟悉的概念被套用上了新的名字。例如，如果在这里列出如下这些关键字：

```
{new, public, this, base, get, set, explicit, unsafe, enum, operator, partial}
```

你当然知道它们是C#的关键字。然而，如果再仔细看看这个集合中的成员，也许会发现尽管它们是C#中的关键字，但是它们有着本质上的语义区别。比如，`enum`这个关键字定义了一个从`System.Enum`派生的类型，`this`和`base`关键字允许分别引用当前的对象或者当前对象的父类。`unsafe`关键字被用来确认一个不可以直接被CLR监控的代码段，`operator`则允许构建一个可以通过特定的C#操作符（例如加号）来访问的隐藏方法。

同C#这样的高级语言完全不同的是，CIL不仅仅定义了一组通用的关键字。而且根据语义上的内涵不同，这些被CIL编译器识别的标记进一步被划分到3个类别中。

- CIL指令（directive）。
- CIL特性（attribute）。
- CIL操作码（opcode）。

每一个类别的CIL标记都通过一个特别的语法来表示，这些标记组织到一起就可以构建出一个有效的.NET程序集。

18.2.1　CIL指令的作用

在CIL的标记集中有这样一组用于描述.NET程序集总体结构的标记，它们被称作CIL指令。CIL指令用于通知CIL编译器如何定义在程序集中用到的命名空间、类型和成员。

CIL指令在语法上使用一个点（.）前缀来表示（例如`.namespace`、`.class`、`.publickeytoken`、`.method`、`.assembly`等）。因此，如果你的*.il文件（CIL代码文件的扩展名）有1个`.namespace`指令和3个`.class`指令，那么CIL编译器将在生成的程序集里产生一个包含有3个.NET类类型的命名空间。

18.2.2　CIL特性的作用

在很多情况下，仅仅CIL指令不足以表示给出的.NET类型或者类型成员。基于这个原因，很多CIL指令可以同CIL特性结合起来使用，CIL特性可以限定应该如何处理一个CIL指令。例如，一个`.class`

指令可以用public特性（建立这个类型的可见性）、extends特性（明确指定这个类型的基类）和implements特性（列出这个类型支持的一系列接口）修饰。

说明 不要混淆“.NET特性”（见第15章）和“CIL特性”，它们是不同的概念。

18.2.3 CIL操作码的作用

一个.NET程序集、命名空间和类型通过CIL的指令和相关的特性来定义后，最后的任务就是提供类型的实现逻辑。这就是CIL操作码的职责范围。沿用底层开发语言的习惯，CIL操作码往往是无法念出来的。例如，如果你需要定义一个字符串变量，你使用的不是一个容易理解的操作名字LoadString，而是ldstr。

老实说，有些CIL操作码是可以直接和C#中的操作符对应起来（例如box、unbox、throw和sizeof）。读者将看到，CIL操作码总是在成员实现的作用域内使用，同CIL指令不同，它们并不需要点前缀。

18.2.4 区别CIL操作码和CIL助记符

正如刚刚说明的，像ldstr这样的操作码被用来实现一个给定类型的成员。不过在现实中，像ldstr这样的标记其实是用来表示真正二进制操作码的CIL助记符。为了说明区别，我们来看一段C#的代码：

```
static int Add(int x, int y)
{
  return x + y;
}
```

用来表示两个数相加的CIL操作码是0x58，用来表示两个数相减的操作码是0x59，在托管堆上分配一个新对象的操作码是0x73。事实上，由JIT编译器来处理的CIL代码无外乎是一堆二进制数据。

幸好，每一个二进制的CIL操作码都对应有一个助记符。例如，可以使用add而不是二进制的0x58，sub而不是0x59，newobj而不是0x73。明白了两者的区别，就可以清楚像ildasm.exe这样的CIL反编译器就是将二进制的操作码翻译成它们所对应的CIL助记符。例如，下面就是ildasm.exe为之前的Add()提供的CIL：

```
.method private hidebysig static int32 Add(int32 x,
 int32 y) cil managed
{
 // 代码大小9（0x9）
.maxstack 2
 .locals init ([0] int32 CS$1$0000)
 IL_0000: nop
 IL_0001: ldarg.0
 IL_0002: ldarg.1
 IL_0003: add
 IL_0004: stloc.0
 IL_0005: br.s    IL_0007
 IL_0007: ldloc.0
 IL_0008: ret
}
```

除非需要构造非常底层的.NET软件（比如自定义的托管编译器），否则不需要关心这些CIL的数字操作码。实际上，当.NET开发人员在说“CIL操作码”时，通常是指那些助记标记而不是底层的那些二进制数值。

18.3　入栈和出栈：CIL 基于栈的本质

像C#这样的高级.NET语言，总是试图尽量隐藏底层的实现。.NET开发一个不太为人注意的方面就是CIL实际上是一个完全以栈为基础的开发语言。回忆我们前面学习的关于集合命名空间（请参看第9章）中Stack<T>类的功能，它用于压一个值入栈，同时也能够将栈顶的值弹出来以供使用。当然，CIL开发人员不是使用Stack<T>类的对象来实现入栈和出栈的，不过从思路上是相通的。

正式地说，在CIL中用来负责这个栈实现的部分叫做*虚拟执行栈*（virtual execution stack）。从下面的介绍中，你将看到CIL提供了一系列操作码来完成压入值到这个栈中，这个过程的术语叫*加载*。同时，CIL也定义了一系列操作码来将栈顶的值移到内存中（例如本地变量），这个过程的术语叫*存储*（store）。

CIL不允许直接访问一个数据，包括本地变量、方法中传入的变量或者类型的字段数据。为了实现访问，必须显式地加载数据到栈中，并在使用时弹出（请留心这一点，因为这能够解释为什么CIL代码看起来有些冗余）。

说明　回忆一下，CIL不会直接执行，而是按需编译。在CIL代码编译的时候，很多实现的冗余可以优化去除。此外，如果为当前项目开启代码优化（使用Visual Studio Project Properties窗口的Build标签），编译器就会移除各种CIL冗余。

下面通过一段简单的没有参数和返回值的函数PrintMessage()，理解CIL是如何应用这个基于栈的处理模型的。在实现这个方法时，我们只将一个本地字符串变量输出到标准的输出流：

```
public void PrintMessage()
{
  string myMessage = "Hello.";
  Console.WriteLine(myMessage);
}
```

如果你观察过C#编译器是如何将这个方法翻译成CIL的，将会首先看到PrintMessage()方法使用.locals指令为本地变量定义了一个存储空间。接着，这个本地字符串通过ldstr（加载字符串）和stloc.0（可以这样理解：存储当前的值到索引为零的本地变量）被加载和存储到这个本地变量中。

这个在索引0处的值接着通过ldloc.0（加载索引为0的局部参数）被加载到内存来供System.Console.WriteLine()方法使用（通过call操作码指定）。最后，这个方法以ret操作码结束返回。下面是PrintMessage()方法的CIL代码（注意，为了简洁，我删除了nop操作码）：

```
.method public hidebysig instance void PrintMessage() cil managed
{
  .maxstack  1
  // 定义一个本地字符串变量（在索引0处）
  .locals init ([0] string myMessage)
```

```
    // 加载一个值为"Hello"的字符串
    ldstr  " Hello."

    // 存储字符串的值到栈上的本地变量
    stloc.0

    // 调用索引0处的值
    ldloc.0

    // 使用当前的值调用方法
    call  void [mscorlib]System.Console::WriteLine(string)
    ret
  }
```

说明 你可以看到，CIL支持使用双斜杠的注释语法（也支持/*...*/这个语法）。同C#一样，注释将被CIL编译器完全忽略。

介绍了CIL指令、特性和操作符基础知识之后，现在来研究CIL编程的实际使用，我们从正反向工程开始讨论。

18.4 正反向工程

读者已经知道可以使用ildasm.exe来查看由C#编译器生成的CIL代码（参见第1章）。不过也许不知道ildasm.exe还允许将加载到ildasm.exe的程序集中的CIL都导出到一个外部文件中。一旦有了CIL代码，就可以使用CIL编译器ilasm.exe任意编辑和重新编译代码。

说明 记得吗，reflector.exe可以用于查看某个程序集的CIL代码，也可以把CIL代码翻译为接近的C#代码。

这个技术叫做正反向工程（round-trip engineering）。在以下这些情况下，它将很有用处。

- 需要修改一个没有源代码的程序集。
- 正在使用的.NET语言编译器不够完美，产生了一些效率不足的CIL代码，而用户希望修改。
- 用户在构造可与COM互操作的库并且希望补充那些在转换过程中丢失的IDL特性，例如COM的[helpstring]特性。

为了解释正反向工程的过程，我们使用文本编辑器来创建一个新的C#代码文件（HelloProgram.cs），并且定义下面的类类型（如果你愿意，也可以使用Visual Studio创建新的控制台项目。但是记住要删除AssemblyInfo.cs这个文件来减少生成的CIL代码数量）：

```
// 简单的C#控制台程序
using System;

// 注意，不能在命名空间中包括类，以简化生成的CIL代码
class Program
{
  static void Main(string[] args)
  {
```

```
      Console.WriteLine("Hello CIL code!");
      Console.ReadLine();
    }
  }
```

将这个文件保存到一个方便的位置（如C:\RoundTrip），然后使用csc.exe编译：

```
csc HelloProgram.cs
```

现在，选择FileDump菜单选项打开带有ildasm.exe的HelloProgram.exe，将原始CIL代码保存到一个新的*.il文件（HelloProgram.il），这个文件位于包含已编译程序集的文件夹中（结果对话框中的所有默认值都保持不变）。

说明　将程序集中的内容转储到文件时，ildasm.exe会生成一个*.res文件。此时，本章所有的源代码文件都可以忽略或删除，因为不需要再用到它们了。

现在，可以使用任意的文本编辑器查看HelloProgram.il。结果如下（有少量的格式更改和注释）：

```
// 引用的程序集
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}

// 我们的程序集
.assembly HelloProgram
{
  /****  为了清楚，删除了TargetFrameworkAttribute数据 ****/

  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module HelloProgram.exe
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003
.corflags 0x00000003

// Program类的定义
.class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    // 标识出这个方法是可执行文件的入口点
    .entrypoint
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr  "Hello CIL code!"
    IL_0006:  call void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  call string [mscorlib]System.Console::ReadLine()
    IL_0011:  pop
```

```
    IL_0012:  ret
  }

  // 默认构造函数
  .method public hidebysig specialname rtspecialname
    instance void  .ctor() cil managed
  {
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  }
}
```

首先需要注意的是，打开的*.il文件声明了编译当前程序集所需要引用的外部程序集。这里，读者可以看到有一个.assembly extern标记用来标识总会出现的mscorlib.dll。当然，读者的类库也许会用到其他程序集的类型，那么就会在这里看到对应的.assembly extern指令。

接着看到的是被赋予了一个默认0.0.0.0版本的HelloProgram.exe程序集的正式定义（如果没有通过[AssemblyVersion]特性来指定一个值的话）。接下来是进一步通过.module、.imagebase这些CIL指令进一步说明该程序集。

在记录了引用的外部程序集和定义了当前的程序集后，定义Program类型。请注意这个.class指令有很多特性（多数是一些可选的特性），例如extends，它标识该类型的基类：

```
.class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
{ ... }
```

其余代码实现了这个类的默认构造函数和Main()方法，都用.method指令定义。一旦成员通过正确的指令和特性定义后，就由操作码来实现。

有一点非常重要，在CIL中与.NET类型（例如System.Console）交互时，总是需要使用这个类型的完全限定名。而且，在这个完全限定名前还需要加上以方括号括起的定义这个类型的程序集的友好名字。考虑下面Main()方法的CIL实现：

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr "Hello CIL code!"
  IL_0006:  call void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  call string [mscorlib]System.Console::ReadLine()
  IL_0011:  pop
  IL_0012:  ret
}
```

下面CIL代码中的默认构造函数使用了另外一个“围绕加载”（load-centric）的操作指令（ldarg.0）。这里，加载到栈中的值不是由我们给出的自定义变量，而是当前的对象引用（下面会进一步说明）。同时也要注意，这个默认的构造函数显式地调用了基类的构造函数，这里就是System.Object：

```
.method public hidebysig specialname rtspecialname
  instance void  .ctor() cil managed
{
  .maxstack  8
```

```
    IL_0000:  ldarg.0
    IL_0001:  call instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  }
```

18.4.1 CIL代码标签的作用

读者一定已经注意到了，在每一行代码前都有一个形如IL_XXXX:的前缀（例如IL_0000:、IL_0001等）。这些标记被称作代码标签（code label），是可以随便修改的（只要在同一个成员作用域中没有重复）。当使用ildasm.exe导出一个程序集时，将会自动在前面加上IL_XXXX这样的代码标签。当然，也可以用更有描述性的方法来标识：

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  .maxstack  8
  Nothing_1:  nop
  Load_String:  ldstr  "Hello CIL code!"
  PrintToConsole: call void [mscorlib]System.Console::WriteLine(string)
  Nothing_2:  nop
  WaitFor_KeyPress: call string [mscorlib]System.Console::ReadLine()
  RemoveValueFromStack:  pop
  Leave_Function:  ret
}
```

事实上，大多数代码标签完全是可选的。只有当我们编写有多个分支和循环结构的CIL代码，通过这些代码标签指定逻辑流转到哪里的时候，这些代码标签才是必需的。对于当前的示例，完全可以全部移除这些自动生成的标签，不会有什么副作用：

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack  8
  nop
  ldstr  "Hello CIL code!"
  call void [mscorlib]System.Console::WriteLine(string)
  nop
  call string [mscorlib]System.Console::ReadLine()
  pop
  ret
}
```

18.4.2 与CIL交互：修改*.il文件

现在，在对基本的CIL文件的组成有所了解的基础上，完成正反向工程之旅。我们的目标是对这个CIL文件做如下修改：

- 增加对System.Windows.Forms.dll程序集的引用；
- 在Main()中增加加载一个局部字符串；
- 调用System.Windows.Forms.MessageBox.Show()方法，使用上面的局部字符串作为参数。

首先通过增加一个新的.assembly指令（用extern特性修饰）来表示你需要使用System.Windows.Forms.dll程序集。我们只需要修改*.il文件，在表示外部引用mscorlib的代码后增加如下逻辑：

```
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
  .ver 4:0:0:0
}
```

要清楚的是，赋给.ver指令的数值可能会根据你所安装的.NET平台版本的不同而不同。在上面，我们使用的是System.Windows.Forms.dll的4.0.0.0版本，它的公钥标记是B77A5C561934E089。如果打开GAC（参照第14章）查到你计算机上的System.Windows.Forms.dll程序集版本，就可以复制正确的版本和公钥标记的值。

下面，需要修改Main()函数。从*.il文件中找到这个函数，删除实现部分（需要保留.maxstack和.entrypoint指令，关于这两个指令的作用后面有详细的说明）：

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 8
  // TODO: 编写新的CIL代码
}
```

重复一下，我们的目的是将一个新的字符串入栈，然后调用MessageBox.Show()方法（而不是原来的Console.WriteLine()方面）。前面提到过，在使用外部定义的类型时，必须使用这个类型的完整名称(同程序集的友好名称结合使用)。也请注意，在CIL中，每个方法调用记录完整的返回类型。将Main()方法修改如下：

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack  8

  ldstr  "CIL is way cool"
  call valuetype [System.Windows.Forms]
    System.Windows.Forms.DialogResult
    [System.Windows.Forms]
    System.Windows.Forms.MessageBox::Show(string)
  pop
  ret
}
```

实际上，上面的CIL代码上对应于如下的C#类定义：

```
class Program
{
  static void Main(string[] args)
  {
    System.Windows.Forms.MessageBox.Show("CIL is way cool");
  }
}
```

18.4.3 使用ilasm.exe编译CIL代码

假设已经修改并保存了这个*.il文件，就可以使用ilasm.exe（CIL编译器）来编译一个新的.NET程序集。这个CIL编译器有大量命令行参数（通过-?选项可以查看它们）。表18-1列出了一些重要的参数。

表18-1　ilasm.exe的命令行参数

参　数	作　用
/debug	包括调试信息（例如本地变量、参数的名字和行号）
/dll	输出*.dll文件
/exe	输出*.exe文件。这个是默认设置，可以忽略
/key	编译程序集时使用给定的*.snk文件强名字
/output	指定输出的文件名和扩展名。如果没有使用此参数，那么产生的文件名（减去文件扩展名）同第一个源文件名相同

在Developer Command提示符中输入以下命令，就可以将修改后的HelloProgram.il文件编译成.NET的*.exe文件了：

```
ilasm /exe HelloProgram.il /output=NewAssembly.exe
```

如果一切都正确，那么将看到一个如下所示的报告：

```
Microsoft (R) .NET Framework IL Assembler.  Version 4.0.21006.1
Copyright (c) Microsoft Corporation.  All rights reserved.
Assembling 'HelloProgram.il'  to EXE --> 'NewAssembly.exe'
Source file is UTF-8

Assembled method Program::Main
Assembled method Program::.ctor
Creating PE file

Emitting classes:
Class 1:        Program

Emitting fields and methods:
Global
Class 1 Methods: 2;
Emitting events and properties:
Global
Class 1
Writing PE file
Operation completed successfully
```

现在，可以运行这个新生成的程序了。这次将看到消息显示在消息框中而不是控制台窗口中。尽管这个简单示例的输出结果没什么大不了的，但它演示了CIL正反向编程的一种实际用法。

18.4.4　peverify.exe的作用

当使用CIL代码编译和修改程序集时，最好能够使用peverify.exe命令行工具来验证编译的二进制映像文件格式上是正确的.NET映像文件。

```
peverify NewAssembly.exe
```

这个工具检查指定的程序集文件中的所有操作码是否是有效的CIL代码。例如，就CIL代码而言，赋值的栈在退出一个函数之前必须是空的。如果忘记弹出任何保存的值，ilasm.exe编译器仍然会生成

一个有效的程序集（因为编译器仅仅关心语法）。另一个方面，peverify.exe关心的是语义。如果忘记在退出函数前清空栈，peverify.exe将会在你试图运行代码库之前通知你。

源代码 RoundTrip文件的源代码位于Chapter 18子目录下。

18.5 CIL 指令和特性

现在已经知道如何使用ildasm.exe，并且使用ilasm.exe来进行正反向工程，我们就可以研究一下CIL自身的语法以及语义。在下面的内容里，我们来共同完成一个包含有多个类型的自定义命名空间。不过，为了简化，这些类型将不包含成员的实现部分。只要掌握了如何创建空类型，就可以进一步考虑如何使用CIL操作码来实现真正意义上的成员了。

18.5.1 在CIL中指定外部引用程序集

使用任意一种编辑器创建一个名为CILTypes.il的文件。首先，列出所有被当前程序集使用到的外部程序集。本例中将只用到mscorlib.dll中定义的类型。为此，需要使用external特性限定.assembly指令。如果引用的是一个强名字程序集，例如mscorlib.dll，还需要指定.publickeytoken和.ver指令：

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
```

说明 严格来说，不需要显式引用mscorlib.dll作为外部引用，因为ilasm.exe会自动加上。但是，对于CIL项目需要的每一个外部.NET库，需要编写类似的.assembly extern指令。

18.5.2 在CIL中定义当前程序集

下一步要做的是使用.assembly指令定义当前这个程序集。在最简单的程度上，一个程序集的定义只需要给出二进制文件名即可：

```
// 我们的程序集
.assembly CILTypes { }
```

在真正意义上定义一个.NET程序集时，还需要在声明部分中增加一些指令。就这个例子而言，可以通过.ver指令来加上一个版本号1.0.0.0（注意每个数字之间是使用冒号来分开的，而不是C#中常见的点）：

```
// 我们的程序集
.assembly CILTypes
{
  .ver 1:0:0:0
}
```

由于CILTypes是一个单文件程序集，可以使用一个.module指令来完成这个程序集的定义，它的作用是给出这个.NET二进制文件的官方名称——CILTypes.dll：

```
.assembly CILTypes
{
 .ver 1:0:0:0
}
// 单文件程序集的模块
.module CILTypes.dll
```

除了.assembly和.module之外，还有对组建中的.NET二进制的整个结构进行进一步配置的CIL指令。表18-2列出了几个常用的程序集级别的指令。

表18-2　其他以程序集为中心的指令

指　　令	作　　用
.mresources	如果你的程序集使用了内部资源（例如位图或者字符串表），这个指令用来给出包含了内嵌资源的文件名称
.subsystem	这个指令用来给出指定的程序集运行时的UI。例如，2表示这个程序集应该以GUI程序的方式运行，而3表示这是一个控制台执行程序

18.5.3　在CIL中定义命名空间

现在，已经定义了程序集的外观以及需要使用的外部程序集引用，接下来可以使用.namespace来创建一个.NET命名空间MyNameSpace：

```
// 我们的程序集有一个命名空间
.namespace MyNamespace {}
```

同C#一样，CIL命名空间定义可以嵌套在其他命名空间内部。这里我们不需要定义根命名空间，但为了说明清楚，假定你想创建一个叫做MyCompany的根命名空间：

```
.namespace MyCompany
{
  .namespace MyNamespace {}
}
```

同C#一样，CIL允许定义一个如下所示的嵌套命名空间：

```
// 定义一个嵌套的命名空间
.namespace MyCompany.MyNamespace{}
```

18.5.4　在CIL中定义类类型

空的命名空间没有什么意义，现在来看看如何使用CIL来定义类类型。读者应该不会感到意外，.class指令就是用来定义一个新类的。不过，这个简单的指令可以和非常多的特性结合使用来实现对类型的完整定义。我们通过增加一个新的公共类MyBaseClass来演示这个指令。正如在C#中一样，如果不显式地给出基类，类型将自动从System.Object派生。

```
.namespace MyNamespace
{
  // 假设以System.Object为基类
```

```
  .class public MyBaseClass {}
}
```

当定义的类类型不是从System.Object继承，而是从其他类继承时，就需要使用extends特性。就算需要引用的类型是定义在同一个程序集里的，CIL也要求使用完整的名字（不过，如果基类型是在同一个程序集里，可以省略这个程序集的友好名字前缀）。因此，下面试图扩展MyBaseClass类的代码会导致编译器错误：

```
// 这段代码是无法编译的
.namespace MyNamespace
{
  .class public MyBaseClass {}

  .class public MyDerivedClass
    extends MyBaseClass {}
}
```

要想正确地定义MyDerivedClass的父类，必须给出MyBaseClass的完整名字：

```
// 好多了
.namespace MyNamespace
{
  .class public MyBaseClass {}

  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass {}
}
```

除了public和extends特性外，一个CIL类的定义还可以使用其他特性来定义类型的可视性、字段的格式等。表18-3给出了一些（但不是全部）可以同.class指令结合使用的特性。

表18-3 同.class指令相结合的特性

特　性	作　用
public、private、nested assembly、nested famandassem、nested family、nested famorassem、nested public和nested private	CIL定义了很多特性，可以用它们来定义一个给定类型的可见性。正如你所见到的，原始CIL提供了比C#要多得多的选项。如果你感兴趣，可以参考ECMA 335
abstract sealed	这两个特性用在.class指令中，分别用来定义一个类为抽象类和封闭类
auto、sequential和explicit	这几个特性用来指示CLR如何为成员分配内存。对于类类型来说，默认标志auto是比较合适的。如果你需要使用平台调用访问非托管的C代码，修改默认值会很有用
extends和implements	这两个特性分别定义一个类型的基类（通过extends）和实现一个类型的接口（通过implements）

18.5.5 在CIL中定义和实现接口

有些古怪的是，在CIL中是使用.class指令定义接口类型的。不过，当.class同interface特性结合使用时，类型被实现为CTS接口类型。一旦定义了接口，就可以使用implements特性把它绑定到一个类类型或者结构类型上：

```
.namespace MyNamespace
{
  // 定义一个接口
  .class public interface IMyInterface {}

  // 一个简单的基类
  .class public MyBaseClass {}

  // My DerivedClass实现IMyInterface，并且扩展MyBaseClass类
  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass
    implements MyNamespace.IMyInterface {}
}
```

说明 extends子句必须在implements子句之前。同样，implements子句可以合并多个用逗号分隔的接口列表。

回顾一下第9章，为了实现接口的继承体系，接口也可以作为其他接口类型的基接口。不过，也许和你想的相反，extends特性不可以用来从接口B派生出接口A。extends特性只能够和一个类型的基类结合使用。如果需要拓展一个接口，可以利用implements特性：

```
// 在CIL中继承一个接口
.class public interface IMyInterface {}

.class public interface IMyOtherInterface
  implements MyNamespace.IMyInterface {}
```

18.5.6 在CIL中定义结构

如果一个类型是从System.ValueType扩展来的，那么.class指令可以用来定义一个CTS结构。同时，.class指令还可以和sealed特性结合使用（注意结构不可以作为其他类型的基类）。如果试图这么做，那么ilasm.exe会报告一个编译器错误。

```
// 结构定义总是密封的
.class public sealed MyStruct
  extends [mscorlib]System.ValueType{}
```

注意CIL提供了一个简便的方法来定义结构类型。如果使用了value特性，定义的新类型会自动从[mscorlib]System.ValueType继承。因此，可以像下面这样定义MyStruct：

```
// 定义一个结构的简便方法
.class public sealed value MyStruct{}
```

18.5.7 在CIL中定义枚举

.NET枚举类型是从System.Enum继承的，而后者是System.ValueType类型（因此也必须是密封的）。当希望用CIL定义一个枚举类型时，可以直接从[mscorlib]System.Enum上扩展：

```
// 一个枚举
.class public sealed MyEnum
  extends [mscorlib]System.Enum{}
```

同结构的定义类似，枚举类型可以使用enum特性简便定义：

```
// 快捷方式定义枚举
.class public sealed enum MyEnum{}
```

我们稍后介绍如何指定枚举的名称/值对。

说明 .NET的基本类型（委托类型）也有专门的CIL表示方法。详细信息请参照第10章。

18.5.8 在CIL中定义泛型

泛型类型在CIL语法中也有特定的表示。在第9章中说过，泛型类型或者泛型成员可以有一个或多个类型参数。例如，List<T>类型有一个参数，而Dictinaty<TKey,TValue>有两个。至于CIL，类型参数的个数通过反勾号（`）加上表示类型参数个数的数值来指定。和C#相似，类型参数的实际值使用尖括号封装。

说明 在大多数键盘上，`字符可以在Tab键上面找到（1键的左边）。

例如，假设希望创建一个List<T>变量，其中T是System.Int32类型的。可以在CIL中按如下所示写代码：

```
// 在C#中：List<int> myInts = new List<int>();
newobj instance void class [mscorlib]
  System.Collections.Generic.List`1<int32>::.ctor()
```

注意泛型类被定义为List`1<int32>，因为List<T>只有一个类型参数。如果我们需要定义Dictionary<string,int>类型的话，就可以按如下所示来做：

```
// 在C#中：Dictionary<string, int> d = new Dictionary<string, int>();
newobj instance void class [mscorlib]
  System.Collections.Generic.Dictionary`2<string,int32>::.ctor()
```

再举一个示例，如果我们的泛型类型将另外一个泛型类型作为参数，就可以如下写CIL代码：

```
// 在C#中：List<List<int>> myInts = new List<List<int>>();
newobj instance void class [mscorlib]
 System.Collections.Generic.List`1<class
  [mscorlib]System.Collections.Generic.List`1<int32>>::.ctor()
```

18.5.9 编译CILTypes.il文件

尽管还没有给定义的类型中加入任何成员和实现代码，不过还是可以编译这个*.il文件到一个.NET DLL程序集（必须是DLL，因为还没有定义Main()方法）。打开命令提示窗口，输入如下所示的命令到ilasm.exe：

```
ilasm /dll CilTypes.il
```

一旦完成了上面这步，就可以使用ildasm.exe打开编译的程序集来查看各个类型的创建。在确认了程序集文件的内容之后，运行peverify.exe来检查它：

```
peverify CilTypes.dll
```

注意，如果所有类型都是空的，可以看到会有一些错误提示。下面是部分输出结果：

```
Microsoft (R) .NET Framework PE Verifier.  Version  4.0.21006.1
Copyright (c) Microsoft Corporation.  All rights reserved.

[MD]: Error: Value class has neither fields nor size parameter. [token:0x02000005]
[MD]: Error: Enum has no instance field. [token:0x02000006]
...
```

要想理解如何来编写有内容的类型，首先要仔细看看CIL的基本数据类型。

18.6　.NET 基础类库、C#和 CIL 数据类型的映射

表18-4说明了.NET基类类型是如何映射到对应的C#关键字上的，以及C#的关键字是如何映射到原始CIL的。同时，表18-4也记录了这些CIL类型的速记常量符号。稍后将会看到，这些常量常常被CIL操作码引用。

表18-4　映射.NET基类类型到C#关键字，映射C#关键字到CIL

.NET 基类类型	C#关键字	CIL表示	CIL速记常量
System.SByte	sbyte	int8	I1
System.Byte	byte	unsigned int8	U1
System.Int16	short	int16	I2
System.UInt16	ushort	unsigned int16	U2
System.Int32	int	int32	I4
System.UInt32	uint	unsigned int32	U4
System.Int64	long	int64	I8
System.UInt64	ulong	unsigned int64	U8
System.Char	char	char	CHAR
System.Single	float	float32	R4
System.Double	double	float64	R8
System.Boolean	bool	bool	BOOLEAN
System.String	string	string	无
System.Object	object	object	无
System.Void	void	void	VOID

说明　System.IntPtr和System.UIntPtr类型映射为本地的int和unsigned int（许多COM互操作和P/Invoke场景都广泛使用了这些类型）。

18.7　在 CIL 中定义类型成员

读者已经知道，.NET类型可以支持多种成员。枚举类型由一系列名称/值的对构成。结构和类则

有构造函数、字段、方法、属性、静态成员等。在本书的前17章中，你已经看到了部分成员的CIL定义，不过这里再重新回顾一下各种成员是如何映射到CIL的。

18.7.1 在CIL中定义数据字段

枚举、结构和类都支持数据字段。在每个子句中都将用到.field指令。举个例子，给MyEnum这个枚举类型框架加入3个名称/值的组合（注意值是使用一对括号来指定的）：

```
.class public sealed enum MyEnum
{
 .field public static literal valuetype
  MyNamespace.MyEnum A = int32(0)
 .field public static literal valuetype
  MyNamespace.MyEnum B = int32(1)
 .field public static literal valuetype
  MyNamespace.MyEnum C = int32(2)
}
```

在.NET System.Enum派生的类型作用域内的字段可以支持static和literal特性，读者可能已经猜到了，设置了这些特性后，字段数据的值可以通过类型来直接访问（例如，MyEnum.A）。

说明 赋到枚举类型的值可以以十六进制的方式（以0x为前缀）给出。

当然，当你希望在类或者结构中定义一个数据成员时，不仅仅可以定义公共静态整型数据。比如，你可以更新MyBaseClass来支持两个私有的对象级别的字段数据：

```
.class public MyBaseClass
{
  .field private string stringField = "hello!"
  .field private int32 intField = int32(42)
}
```

在C#中，类成员数据可以被自动赋予默认的值。如果你希望能够自定义对象的初始值，那么就必须创建自定义的构造函数。

18.7.2 在CIL中定义类型的构造函数

CTS支持对象实例化层次和类层次（静态）构造函数。就CIL而言，实例化层次的构造函数使用.ctor来表示，而静态的构造函数通过.cctor（类构造函数）来表示。这两个CIL标记必须要和rtspecialname（返回类型的指定名字）和specialname特性结合才可以使用。简单地说，这些特性用于标识出根据所给的.NET语言进行特别处理的CIL标记。举个例子，在C#中，构造函数不可以有返回类型；不过，在CIL中，构造函数的返回值实际上是void：

```
.class public MyBaseClass
{
  .field private string stringField
  .field private int32 intField

  .method public hidebysig specialname rtspecialname
    instance void .ctor(string s, int32 i) cil managed
  {
```

```
    // TODO:加入实现代码
  }
}
```

注意，.ctor指令是和instance特性结合使用的（因为它不是一个静态的构造函数）。上面的cil managed特性标识出这个方法包含的是CIL代码而不是非托管代码，在平台发出请求期间才可使用非托管代码。

18.7.3　在CIL中定义属性

属性和方法也有CIL的标识。继续我们的例子，给MyBaseClass添加一个公共属性TheString，需要如下所示的代码（注意，这里再次用到了specialname特性）：

```
.class public MyBaseClass
{
...
  .method public hidebysig specialname
    instance string  get_TheString() cil managed
  {
    // TODO: 加入实现代码
  }

  .method public hidebysig specialname
    instance void  set_TheString(string 'value') cil managed
  {
    // TODO: 加入实现代码
  }

  .property instance string TheString()
  {
    .get instance string
      MyNamespace.MyBaseClass::get_TheString()
    .set instance void
      MyNamespace.MyBaseClass::set_TheString(string)
  }
}
```

就CIL而言，属性实际上是映射到以get_和set_为前缀的方法对。.property指令使用相关的.get和.set指令将属性语法映射到正确的、"被特殊命名的"方法上。

说明　注意属性的set方法的传入参数放在单引号中，表示在方法作用域中赋值操作符的右边使用的标识的名字。

18.7.4　定义成员参数

简而言之，在CIL中定义参数的方法和在C#中大同小异。例如，每一个定义的参数都要给出数据类型和参数名称。此外，同C#一样，CIL也要定义输入、输出和按引用传递的参数。同样，CIL还允许定义参数数组（即C#中的params关键字）以及可选参数。

为了说明使用原始CIL定义参数的过程，先假设你需要构造一个方法，它可传递值int32、引用的int32、[mscorlib]System.Collection.ArrayList以及int32类型的单输出参数。在C#中，这样的一个

方法表示如下：

```
public static void MyMethod(int inputInt,
  ref int refInt, ArrayList ar, out int outputInt)
{
  outputInt = 0;  // 仅仅为了让C#编译器通过编译
}
```

如果要把这个方法映射到CIL语法上，C#的引用参数是在数据类型的后面加上&符号（int32&）。

输出参数也是使用一样的方式，只是又加上了CIL的[out]标记。要注意的是，如果一个参数是引用类型（这个例子中是[mscorlib]System.Collections.ArrayList类型），需要加上一个class标记的前缀（不过不要和.class指令混淆）。

```
.method public hidebysig static void MyMethod(int32 inputInt,
  int32& refInt,
  class [mscorlib]System.Collections.ArrayList ar,
  [out] int32& outputInt) cil managed
{
  ...
}
```

18.8 剖析 CIL 操作码

在这一章里，最后要学习的就是各种CIL操作码的作用。回忆一下，操作码就是一种简单的CIL标记，用于构造指定成员的实现逻辑部分。完整的CIL操作码集合可以被划分到下面3个大的类别中。

- 控制操作流程的操作码。
- 求值表达式的操作码。
- （通过参数、局域变量等）访问内存值的操作码。

为了深入了解通过CIL的成员实现，表18-5根据功能的相关性列出了一些逻辑实现中常用的操作码。

表18-5 一些实现相关的CIL操作码

操 作 码	作 用
add、sub、mul、div和rem	用于加减乘除两个数值（rem返回除法操作的余数）
and、or、not和xor	用于在两个值上进行二进制操作
ceq、cgt和clt	用不同的方法比较两个在栈上的值。例如：ceq用于比较是否相等，cgt用于比较是否大于，clt用于比较是否小于
box和unbox	在引用类型和值类型之间转换
ret	退出方法和返回一个值（如果需要）
beq、bgt、ble、blt和switch	控制方法中的条件分支。例如：beq用于表示如果相等就中止到代码标签，bgt用于表示如果大于就中止到代码标签，ble用于表示如果小于等于就中止到代码标签，blt用于表示如果小于就中止到代码标签。所有的分支控制操作码都需要你给出一个CIL代码标签作为条件为真的跳转目的
call	调用一个给定类型的成员
newarr和newobj	在内存中创建一个新的数组或者新的对象类型

下一个CIL操作码（表18-6列出了其中一部分）的类别用于加载（压入）参数到虚拟执行栈中。你会看到这些加载相关的操作码使用ld（加载）的前缀。

表18-6　主要的CIL栈操作码

操 作 码	作　用
ldarg（及多个变化形式）	加载方法的参数到栈中。除了泛型ldarg（需要一个索引作为参数），还有其他很多的变化形式。例如，有一个数字后缀的ldarg操作码（如ldarg_0）来指定需要加载的参数。同时，还有很多ldarg的变化形式允许使用表18-4所示的CIL常量加载指定的数据类型（ldarg_I4，加载int32）和值（ldarg_I4_5加载一个值为5的int32）
ldc（及多个变化形式）	加载一个常数值到栈中
ldfld（及多个变化形式）	加载一个对象实例的成员到栈中
ldloc（及多个变化形式）	加载一个本地变量到栈中
ldobj	获得一个堆对象的所有数据，并将它们放置在栈中
ldstr	加载一个字符串数据到栈中

除了上面这些加载相关的操作码，CIL还提供了很多用于显式弹出栈顶数据的操作。在本章最开始的例子中可以看到，弹出一个栈中的值一般需要存储这个值到一个临时的本地存储空间中以便接下来使用（比如一个参数被方法调用）。了解了这一点，可以看到有很多操作符使用st（存储）前缀用来从虚拟执行栈中弹出当前的值。表18-7列出了一些。

表18-7　弹出操作码

操 作 码	作　用
pop	删除当前栈顶的值，但是并不影响存储值
starg	存储栈顶的值到给出方法的参数，根据索引确定这个参数
stloc（及多个变化形式）	弹出当前栈顶的值并且存储在一个本地变量列表中，根据索引确定这个参数
stobj	从栈中复制一个特定的类型到指定的内存地址
stsfld	用从栈中获得的值替换静态成员的值

要注意的是，这些CIL操作码将不显式地弹出栈中的值来执行当前任务。例如，如果你试图使用sub操作码对两个数字进行减法操作，那么需要清楚的是sub在执行操作前会自动弹出两个有效的值。一旦运算操作完成了，计算的结果将被入栈。

18.8.1　.maxstack指令

当编写原始CIL代码时，需要注意一个特别的指令.maxstack。正如这个名字所说明的，.maxstack确定一个在方法执行阶段可以被压入栈中的最大变量数目。好在.maxstack的默认值是8，对于绝大多数方法来说都应该是足够了。不过，如果你希望明确地指出，可以自行计算栈中的局域变量，并显式地给出这个值：

```
.method public hidebysig instance void
  Speak() cil managed
{
  // 在这个方法的有效范围内，只有值1（字符串字面量）是在栈中
  .maxstack 1
  ldstr "Hello there..."
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

18.8.2 在CIL中声明本地变量

先看一下如何声明一个本地变量。假设你想构造一个没有参数、返回void的MyLocalVariables()方法。在这个方法中，你想定义3个本地变量，数据类型分别是System.String、System.Int32和System.Object。在C#中，可以像下面这样写代码（回忆一下，本地变量在使用前必须要初始化）：

```
public static void MyLocalVariables()
{
  string myStr = "CIL code is fun!";
  int myInt = 33;
  object myObj = new object();
}
```

如果是在CIL中直接定义这个MyLocalVariables()，代码如下：

```
.method public hidebysig static void
  MyLocalVariables() cil managed
{
  .maxstack 8
 // 定义3个本地变量
  .locals init ([0] string myStr, [1] int32 myInt, [2] object myObj)
  // 加载字符串到虚拟执行栈中
  ldstr "CIL code is fun!"
 // 弹出当前的值，并存入本地变量[0]
  stloc.0

 // 加载常量到类型i4（int32的简写），设置值为33
  ldc.i4 33
 // 弹出当前的值，并存入本地变量[1]
  stloc.1

  // 创建一个新对象并放在栈上
  newobj instance void [mscorlib]System.Object::.ctor()
 // 弹出当前的值，并存入本地变量[2]
  stloc.2
  ret
}
```

你可以看到，在原始CIL中分配本地变量，首先是使用.locals指令和init特性。在对应的作用域内，将每一个变量同给出的索引相互关联（例如这里的[0]、[1]和[2]）。可见，每一个索引都由数据类型和可选的变量名称来表示。一旦一个本地变量被定义后，就可以加载值到栈中（使用加载相关的操作码）并且存储本地变量中的值（使用存储相关的操作码）。

18.8.3　在CIL中映射参数到本地变量

我们已经学习了如何使用.local初始指令在原始CIL中声明局域变量，不过还没有看到如何映射传入参数到局部方法中。参考如下的C#静态方法：

```
public static int Add(int a, int b)
{
  return a + b;
}
```

这个看起来很简单的函数如果要用CIL实现，还是有很多讲究的。首先，传入的参数a和b必须使用ldarg压入虚拟的执行栈中。接着，使用add操作码从栈中弹出这两个值，然后计算其和，并将计算出来的结果存回到栈中。最后，这个和被弹出栈并且通过ret操作码返回到调用者。如果使用ildasm.exe来反编译这个C#函数，会看到很多由csc.exe注入的额外标记，不过最关键的CIL其实很简单：

```
.method public hidebysig static int32  Add(int32 a,
  int32 b) cil managed
{
  .maxstack  2
  ldarg.0    // 加载"a"到栈中
  ldarg.1    // 加载"b"到栈中
  add        // 求和
  ret
}
```

18.8.4　this隐式引用

注意上面的例子中，考虑到虚拟执行栈索引是以0开始的，CIL代码使用传入的a和b的索引0和1来引用。

需要始终记住的是，在使用原始CIL代码时，任何非静态函数在接收传入参数的时候都自动隐式地接收了一个附加参数。这个参数就是当前对象的引用（联想C#的this关键字）。考虑到这点，如果Add()方法是非静态的：

```
// 非静态
public int Add(int a, int b)
{
  return a + b;
}
```

那么传入的a和b就要使用ldarg.1和ldarg.2而不是前面的ldarg.0和ldarg.1。原因就是0已经被this引用占用了。参考下面的伪代码：

```
// 这个仅仅是伪代码
.method public hidebysig static int32 AddTwoIntParams(
  MyClass_HiddenThisPointer this, int32 a, int32 b) cil managed
{
  ldarg.0   // 加载MyClass_HiddenThisPointer到栈上
  ldarg.1   // 加载"a"到栈上
  ldarg.2   // 加载"b"到栈上
...
}
```

18.8.5 在CIL中使用循环结构

在C#中，循环结构是使用for、foreach、while和do这些关键字来实现的。每一个在CIL中都有对应的表示。例如下面的for循环：

```
public static void CountToTen()
{
  for(int i = 0; i < 10; i++)
    ;
}
```

现在，你也许会回想起前面提到过的br操作码（br、blt等）可以根据条件控制流程。在这里，可以为for循环设置一个条件，当本地变量i等于或大于10的时候就跳出循环。每产生一次迭代，i的值都要增加1，并且重新检查设置的条件是否满足。

前面提到过，当使用CIL分支操作码时，需要定义一个（或两个）具体的代码标签作为条件为真时的跳出位置。参考一下如下通过ildasm.exe生成的CIL代码（包含自动生成的代码标签）：

```
.method public hidebysig static void CountToTen() cil managed
{
  .maxstack  2
  .locals init ([0] int32 i)  // 初始化本地变量 'i'
  IL_0000:  ldc.i4.0          // 加载这个值到栈中
  IL_0001:  stloc.0           // 存储这个值到索引 '0'
  IL_0002:  br.s IL_0008      // 跳到IL_0008
  IL_0004:  ldloc.0           // 加载索引0的值
  IL_0005:  ldc.i4.1          // 加载值'1'
  IL_0006:  add               // 增加当前在索引0的值
  IL_0007:  stloc.0
  IL_0008:  ldloc.0           // 加载在索引'0'的值
  IL_0009:  ldc.i4.s   10     // 加载'10'到栈上
  IL_000b:  blt.s IL_0004     // 小于?如果是，跳到IL_0004
  IL_000d:  ret
}
```

简单地说，这段CIL代码开始先定义了本地的int32，加载到栈上。之后，就在代码IL_0008到IL_0004之间循环。每次都弹出i的值（以1递增）并且查看i是否比10小。如果是，则退出代码。

源代码 CilTypes示例的源代码位于Chapter 18子目录下。

18.9 使用 CIL 构建.NET 程序集

现在你已经大概了解了原始CIL的语法和语义，那么该使用ilasm.exe和任一文本编辑器构建一个.NET应用程序，来复习和巩固学习成果了。这个应用程序有一个私有部署的单文件*.dll，它包含两个类类型定义和一个与这些类型进行交互的、基于控制台的*.exe。

18.9.1 构建CILCars.dll

首先需要构建一个*.dll来供我们的程序使用。使用一个文本编辑器来创建一个新的*.il文件，命名

为CILCars.il。这个单文件程序集需要使用两个外部的.NET二进制文件。下面是这段CIL的代码：

```
// 引用mscorlib.dll和System.Windows.Forms.dll
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 4:0:0:0
}

// 定义这个单文件程序集
.assembly CILCars
{
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module CILCars.dll
```

这个程序集将包含两个类类型。第一个类型是CILCar，定义两个字段数据（简便起见本例使用了public字段）和一个自定义构造函数。第二个类型是CarInfoHelper，定义一个静态方法Display-CarInfo()，它以CILCar作为参数并且返回void。两个类型都在CILCars这个命名空间中。CILCar的CIL实现如下：

```
// 实现CILCars.CILCar类
.namespace CILCars
{
  .class public auto ansi beforefieldinit CILCar
    extends [mscorlib]System.Object
  {
    // CILCar的数据字段
    .field public string petName
    .field public int32 currSpeed

    // 自定义构造函数允许调用者给数据字段赋值
    .method public hidebysig specialname rtspecialname
        instance void .ctor(int32 c, string p) cil managed
    {
      .maxstack  8

      // 加载第一个参数到栈上并调用基类的构造函数
      ldarg.0  // "this"对象，不是int32!
      call instance void [mscorlib]System.Object::.ctor()

      // 加载第一个和第二个参数到栈上
      ldarg.0  //  "this" 对象
      ldarg.1  // int32参数

      // 存储栈顶的成员(int32)到currSpeed字段
      stfld int32 CILCars.CILCar::currSpeed

      // 加载字符串参数并且存储到petName字段
      ldarg.0  // "this" 对象
      ldarg.2  // string 参数
      stfld string CILCars.CILCar::petName
```

```
      ret
    }
  }
}
```

需要始终记得的是，非静态成员的第一个参数是当前的对象引用，CIL的第一段代码就是加载这个对象引用并且调用基类的构造函数。接下来，将传入的构造函数参数入栈并且使用stfld操作码(store in field)存储到类型的字段数据中。

接下来，需要在这个命名空间中实现第二个类型：CILCarInfo。它的主要作用可以在静态的Display()方法中看到。简单地说，这个方法的作用就是接收传入的CILCar参数，提取其字段数据的值并在Windows窗体消息框上显示它。下面是CILCarInfo（它定义在CILCars命名空间中）的完整实现：

```
.class public auto ansi beforefieldinit CILCarInfo
  extends [mscorlib]System.Object
{
  .method public hidebysig static void
    Display(class CILCars.CILCar c) cil managed
  {
    .maxstack  8

    // 需要一个局部的字符串变量
    .locals init ([0] string caption)

    // 加载字符串和传入的CILCar到栈上
    ldstr "{0}'s speed is:"
    ldarg.0

    // 将CILCar的petName放到栈上，然后调用静态的String.Format()方法
    ldfld string CILCars.CILCar::petName
    call string [mscorlib]System.String::Format(string, object)
    stloc.0

    // 现在加载currSpeed字段的值，取得它的字符串表示（调用ToString()）
    ldarg.0
    ldflda int32 CILCars.CILCar::currSpeed
    call instance string [mscorlib]System.Int32::ToString()
    ldloc.0

    // 现在调用MessageBox.Show()方法并传入加载的值
    call valuetype [System.Windows.Forms]
          System.Windows.Forms.DialogResult
          [System.Windows.Forms]
          System.Windows.Forms.MessageBox::Show(string, string)
    pop
    ret
  }
}
```

尽管上面的CIL代码比看到的CILCar的实现要多些，不过逻辑上比较简单。首先，由于是定义一个静态方法，不需要关心隐式的对象引用（因此，ldarg.0操作码实际上加载的是传入的CILCar参数）。

最开始是加载一个字符串（"{0}'s speed is"）到栈上，接着的是CILCar参数。一旦这两个值安排妥当，加载petName字段的值，并调用静态的System.String.Format()方法来用CILCar里的昵称代替花括号占位符。

当处理currSpeed字段时，采用类似的过程，不过这次使用ldflda操作码来加载这个参数地址到栈上。然后调用System.Int32.ToString()来将该地址的值转换为字符串类型。最后，一旦两个字符串都已经格式化好了，就调用MessageBox.Show()方法。

现在，就可以使用ilasm.exe和如下命令来编译新的*.dll：

```
ilasm /dll CILCars.il
```

并使用peverify.exe来验证CIL：

```
peverify CILCars.dll
```

18.9.2 构建CILCarClient.exe

现在可以使用Main()方法构建一个简单的*.exe程序集。

- 构建一个CILCar对象。
- 把这个对象作为参数传送给静态方法CILCarInfo.Display()。

创建一个新的*.il文件CarClient.il，并定义外部引用mscorlib.dll和CILCars.dll（不要忘记把这个.NET程序集复制一份到客户端的程序目录中）。然后定义一个操作CILCars.dll程序集的类型Program。下面是完整的代码：

```
// 外部程序集引用
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
  .ver 4:0:0:0
}
.assembly extern CILCars
{
  .ver 1:0:0:0
}

// 我们的执行程序集
.assembly CarClient
{
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module CarClient.exe

// Program类型的实现
.namespace CarClient
{
  .class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
  {
    .method private hidebysig static void
    Main(string[] args) cil managed
    {
      // 为*.exe的入口点做标记
      .entrypoint
      .maxstack  8

      // 声明局部CILCar变量并为栈上的ctor请求输入值
      .locals init ([0] class
```

```
      [CILCars]CILCars.CILCar myCilCar)
      ldc.i4 55
      ldstr "Junior"

      // 构建新的CilCar；存储并加载引用
      newobj instance void
        [CILCars]CILCars.CILCar::.ctor(int32, string)
      stloc.0
      ldloc.0

      // 调用Display()并传入栈顶值
      call void [CILCars]
         CILCars.CILCarInfo::Display(
             class [CILCars]CILCars.CILCar)
      ret
    }
  }
}
```

需要强调的一个操作码是.entrypoint。回忆一下我们前面的讨论，这个操作码是用做标识*.exe函数的一个方法作为模块的入口。实际上，由于.entrypoint就是CLR用来标识初始执行方法的，所以该方法也可以是Main()之外的其他任何方法。其余的在Main()方法中的CIL代码主要执行一些入栈和出栈操作。

但是一定要记住，CILCar的创建涉及.newobj操作码的使用。另外，回想一下，要使用原始CIL调用类型成员时，需要利用双冒号语法，还一定要使用类型的完全限定名。接下来就可以使用ilasm.exe来编译新文件，使用peverify.exe来验证程序集，最后执行程序。在命令提示符中输入以下命令：

```
ilasm CarClient.il
peverify CarClient.exe
CarClient.exe
```

源代码 CilCars示例的源代码位于Chapter 18子目录下。

18.10 动态程序集

也许你已经注意到了，使用CIL代码来开发一个复杂的.NET应用程序将会是一个非常繁重的体力劳动。一方面，CIL是一种极端自我表达的开发语言，能够允许你和任何一种被CTS支持的语言交互。另外一方面，开发原始CIL代码也是乏味的，容易出错，令人痛苦。当然，这个知识本身还是重要的，你也许会想知道牢记CIL的语法规则究竟有多重要。答案是，具体情况具体分析。当然，大多数.NET编程不需要查看、编辑或者编写原始CIL代码。不过，有了CIL的基本知识，现在就可以研究和探索动态程序集（同静态程序集相对应）以及System.Reflection.Emit命名空间的作用。

读者的第一个问题也许是："动态和静态程序集到底有什么区别？"根据定义，静态程序集是.NET直接从磁盘存储器加载的.NET二进制文件，也就是说在CLR请求加载它们的时候，它们是在硬盘上的一些物理文件（也可能是多文件程序集中的一组文件）。正如读者所料，每次编译C#源代码，都会生成一个静态程序集。

动态程序集在运行中通过使用System.Reflection.Emit命名空间提供的类型在内存中创建。

System.Reflection.Emit命名空间使得在运行时创建程序集及其模块、类型定义以及CIL实现逻辑成为可能。一旦完成了这些，就可以将内存中的二进制程序保存到磁盘上生成一个新的静态程序集。当然，使用System.Reflection.Emit构造动态程序集需要对CIL操作码有一定程度的理解。

尽管创建动态程序集是一种相当高级的编程技术，也不是很常见，不过在一些情况下，这个技术非常有用。

- 构建需要根据用户输入来生成程序集文件的.NET开发工具。
- 构建需要在运行时通过元数据来生成远程类型的代理的程序。
- 希望加载静态程序集并能够动态插入新的类型到二进制图像中。

那么，现在来看看System.Reflection.Emit这个命名空间中的类型。

.NET运行时引擎的几个方面包括在后台动态生成程序集。例如，ASP.NET使用这个技术来把标记和服务器端脚本映射到运行时对象模型。LINQ也根据各种查询表达式来生成代码。现在来看一下System.Reflection.Emit中的一些类型。

18.10.1　System.Reflection.Emit命名空间

创建动态程序集需要对CIL操作码有一定程度的了解，不过System.Reflection.Emit命名空间提供的类型则尽可能地隐藏了CIL的复杂度。例如，可以使用TypeBuilder类而不是直接使用CIL指令和特性来定义一个类。如果希望定义一个实例级的构造函数，不需要使用specialname、rtspecialname或者.ctor标记，而是使用ConstructorBuilder。表18-8记录了System.Reflection.Emit命名空间的一些重要成员。

表18-8　System.Reflection.Emit命名空间的一些成员

成　员	作　用
AssemblyBuilder	运行时创建程序集文件（*.dll或者*.exe）。*.exe必须调用ModuleBuilder.SetEntryPoint()来设置模块的入口函数。如果没有指定入口函数，那么就会生成*.dll文件
ModuleBuilder	定义当前程序集中的模块集
EnumBuilder	创建.NET枚举类型
TypeBuilder	运行时创建模块中的类、接口、结构、委托
MethodBuilder LocalBuilder PropertyBuilder FieldBuilder ConstructorBuilder CustomAttributeBuilder ParameterBuilder EventBuilder	运行时创建类型成员（比如方法、本地变量、属性、构造函数以及特性）
ILGenerator	产生CIL操作码到给定的类型成员
OpCodes	提供了很多可以映射到CIL操作码上的成员。这个类型需要同System.Reflection.Emit.ILGenerator提供的成员结合使用

总而言之，System.Reflection.Emit命名空间提供的类型允许你在构建动态程序集时用编程方式表示原始的CIL标记。你会看到下面的代码中用到了很多这些成员。不过ILGenerator值得我们先来看看。

18.10.2 System.Reflection.Emit.ILGenerator的作用

顾名思义，ILGenerator类型用来注入CIL操作码到一个给定的类型成员。因为这个类型没有公共构造函数，所以一般而言，不需要直接创建ILGenerator对象，而是通过调用一些围绕构造的类型（例如MethodBuilder和ConstructorBuilder）的指定方法来获得对ILGenerator类型的有效引用。例如：

```
// 从ConstructorBuilder对象'myCtorBuilder'获得引用
   ConstructorBuilder myCtorBuilder =
  new ConstructorBuilder(/* ...various args... */);

ILGenerator myCILGen = myCtorBuilder.GetILGenerator();
```

一旦得到了ILGenerator的引用，就可以使用它提供的各种方法生成原始CIL操作码。表18-9列出了ILGenerator的部分（但不是全部）方法。

表18-9 ILGenerator的部分方法

方　　法	作　　用
BeginCatchBlock()	开始一个catch程序块
BeginExceptionBlock()	开始一个没有过滤的异常捕获块
BeginFinallyBlock()	开始一个finally块
BeginScope()	开始一个词汇范围
DeclareLocal()	定义一个本地变量
DefineLabel()	定义一个新标签
Emit()	被重载多次以生成CIL操作码
EmitCall()	压入一个call或者callvirt操作码到CIL流
EmitWriteLine()	根据不同类型的值，产生一个对Console.WriteLine()的调用
EndExceptionBlock()	结束一个异常程序块
EndScope()	结束一个词汇范围
ThrowException()	产生抛出异常的指令
UsingNamespace()	指定用来对本地变量求值的命名空间，并监控当前活动的程序块

ILGenerator的关键方法是同System.Reflection.Emit.OpCodes类类型联合使用的Emit()。正如前面提到的，System.Reflection.Emit.OpCodes公开了很多只读的、映射到原始CIL操作码的字段。可以从在线帮助查看相关的例子程序，或者查看完整的成员定义。

18.10.3 产生动态的程序集

让我们通过创建一个叫做MyAssembly.dll的单文件动态程序集来说明如何在运行时动态创建.NET程序集。这个模块包含一个名为HelloWorld的类。HelloWorld类支持一个默认的构造函数和一个用来

给string类型的私有成员变量（theMessage）赋值的自定义构造函数。此外，HelloWorld还支持一个公共的实例方法SayHello()，这个方法可以输出一个欢迎信息到标准I/O流中；HelloWorld还支持一个实例方法GetMsg()，可以返回内部的私有字符串。实际上，是通过编程生成如下的类类型：

```
// 这个类将被System.Reflection.Emit在运行时创建
public class HelloWorld
{
  private string theMessage;
  HelloWorld() {}
  HelloWorld(string s) {theMessage = s;}

  public string GetMsg() {return theMessage;}
  public void SayHello()
  {
    System.Console.WriteLine("Hello from the HelloWorld class!");
  }
}
```

假设你已经创建了一个叫做DynamicAsmBuilder的新Visual Studio控制台应用项目。引入System.Reflection、System.Reflection.Emit和System.Threading命名空间，并且定义一个静态方法CreateMyAsm()。这个方法负责：

- 定义动态程序集的特征（名字、版本等）;
- 实现HelloClass类型；
- 保存内存中的程序集到一个物理文件。

CreateMyAsm()方法只有一个System.AppDomain类型的参数，通过它可以访问当前应用程序域的AssemblyBuilder类型（参考第16章关于.NET应用程序域的讨论）。下面是完整的代码，随后进行分析：

```
// 调用者传入一个AppDomain类型
public static void CreateMyAsm(AppDomain curAppDomain)
{
  // 建立通用的程序集特征
  AssemblyName assemblyName = new AssemblyName();
  assemblyName.Name = "MyAssembly";
  assemblyName.Version = new Version("1.0.0.0");

  // 在当前AppDomain（应用程序域）中创建一个新的程序集
  AssemblyBuilder assembly =
    curAppDomain.DefineDynamicAssembly(assemblyName,
    AssemblyBuilderAccess.Save);

  // 鉴于我们构造的是一个单文件程序集，模块的名字就是程序集的名字
  ModuleBuilder module =
    assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");

  // 定义一个公共类"HelloWorld"
  TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
    TypeAttributes.Public);

  // 定义一个私有字符串成员变量"theMessage"
  FieldBuilder msgField =
    helloWorldClass.DefineField("theMessage", Type.GetType("System.String"),
    FieldAttributes.Private);

  // 创建自定义的构造函数
  Type[] constructorArgs = new Type[1];
```

```
  constructorArgs[0] = typeof(string);
  ConstructorBuilder constructor =
    helloWorldClass.DefineConstructor(MethodAttributes.Public,
    CallingConventions.Standard,
    constructorArgs);
  ILGenerator constructorIL = constructor.GetILGenerator();
  constructorIL.Emit(OpCodes.Ldarg_0);
  Type objectClass = typeof(object);
  ConstructorInfo superConstructor =
    objectClass.GetConstructor(new Type[0]);
  constructorIL.Emit(OpCodes.Call, superConstructor);
  constructorIL.Emit(OpCodes.Ldarg_0);
  constructorIL.Emit(OpCodes.Ldarg_1);
  constructorIL.Emit(OpCodes.Stfld, msgField);
  constructorIL.Emit(OpCodes.Ret);

  // 创建默认构造函数
  helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);
  // 创建GetMsg()方法
  MethodBuilder getMsgMethod =
    helloWorldClass.DefineMethod("GetMsg", MethodAttributes.Public,
    typeof(string), null);
  ILGenerator methodIL = getMsgMethod.GetILGenerator();
  methodIL.Emit(OpCodes.Ldarg_0);
  methodIL.Emit(OpCodes.Ldfld, msgField);
  methodIL.Emit(OpCodes.Ret);

  // 创建SayHello方法
  MethodBuilder sayHiMethod =
    helloWorldClass.DefineMethod("SayHello",
    MethodAttributes.Public, null, null);
  methodIL = sayHiMethod.GetILGenerator();
  methodIL.EmitWriteLine("Hello from the HelloWorld class!");
  methodIL.Emit(OpCodes.Ret);

  // 创建类HelloWorld (Baking是创建一个类型的正式术语)
  helloWorldClass.CreateType();

  // 将这个程序集保存到文件(可选)
  assembly.Save("MyAssembly.dll");
}
```

18.10.4 产生程序集和模块集

上面的方法先使用AssemblyName和Version类型（在System.Reflection命名空间里定义）构造了一个程序集的最小特性集合。接下来，可以通过实例级的AppDomain.DefineDynamicAssembly()方法获得AssemblyBuilder类型（回想一下，调用方法将一个AppDomain引用传入CreateMyAsm（ ）方法中）：

```
// 定义一般程序集特性，并获得对AssemblyBuilder的访问
public static void CreateMyAsm(AppDomain curAppDomain)
{
  AssemblyName assemblyName = new AssemblyName();
  assemblyName.Name = "MyAssembly";
  assemblyName.Version = new Version("1.0.0.0");

  // 使用当前的AppDomain创建一个新的程序集
  AssemblyBuilder assembly =
```

```
    curAppDomain.DefineDynamicAssembly(assemblyName,
    AssemblyBuilderAccess.Save);
  ...
}
```

可见，当调用AppDomain.DefineDynamicAssembly()时，必须指定要定义的程序集的访问模式，这些访问模式中最常用的值列在了表18-10中。

表18-10　常用的AssemblyBuilderAccess枚举值

值	作　用
ReflectionOnly	表示一个动态程序集只能够通过反射访问
Run	表示一个动态程序集可以在内存执行，但是不能够保存到硬盘
RunAndSave	表示一个动态程序集可以在内存中执行也可以保存到硬盘上
Save	表示一个动态程序集可以保存到硬盘但是不可以在内存中执行

下一个任务就是为这个新的程序集定义模块集。考虑到这个程序集只有一个文件，只需要定义一个模块即可。如果使用DefineDynamicModule()方法构造多文件程序集，还要通过可选的第二个参数来表示要操作的模块的名称（如myMod.dotnetmodule）。当创建单文件程序集时，模块的名字和程序集的名字相同。在任何情况下，只要DefineDynamicModule()方法返回，就可以得到一个ModuleBuilder类型的引用。

```
// 单文件程序集
ModuleBuilder module =
  assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");
```

18.10.5　ModuleBuilder类型的作用

ModuleBuilder是开发动态程序集的关键类型。正如读者所期望的，ModuleBuilder支持一系列成员方法，可用来定义模块包含的各种类型（类、接口和结构等）和嵌入资源（字符串表、图像等）。表18-11介绍了部分用作创建的方法。（这些方法会返回需要构造的相关类型。）

表18-11　ModuleBuilder类型的部分成员

方　法	作　用
DefineEnum()	产生.NET枚举类型定义
DefineResource()	产生存储在这个模块中的托管的嵌入资源
DefineType()	构造一个TypeBuilder，可以用来定义值类型、接口和类类型（包括委托）

ModuleBuilder类的关键成员就是DefineType()。除了通过简单字符串指定类型名称，还可以使用System.Reflection.TypeAttributes枚举来进一步描述类型的格式。表18-12列出了TypeAttributes枚举的部分（但不是全部）关键成员。

表18-12　TypeAttributes枚举的部分成员

成　员	作　用
Abstract	类型是抽象的
Class	类型是类

（续）

成 员	作 用
Interface	类型是接口
NestedAssembly	具有程序集访问级别的嵌套类，只能被所在的程序集方法访问
NestedFamAndAssem	具有程序集访问级别和成员访问级别的嵌套类，因此只能够被属于成员和程序集交集的方法访问
NestedFamily	具有成员访问级别的嵌套类，只能够被本类型及子类型的方法访问
NestedFamORAssem	具有程序集访问级别和成员访问级别的嵌套类，能够被属于成员和程序集并集的方法访问
NestedPrivate	私有访问级别的嵌套类
NestedPublic	公有访问级别的嵌套类
NotPublic	非公有的类
Public	公有的类
Sealed	具体的不可扩展的类
Serializable	可以序列化的类

18.10.6 产生HelloClass类型和字符串成员变量

现在已经了解了ModuleBuilder.CreateType()方法的作用，下面看一下如何使用它来产生公共的HelloClass类类型和私有的字符串变量：

```
// 定义一个公共类MyAssembly.HelloWorld
TypeBuilder helloWorldClass = module.DefineType("MyAssembly.HelloWorld",
  TypeAttributes.Public);

// 定义私有字符串成员变量theMessage
FieldBuilder msgField =
  helloWorldClass.DefineField("theMessage",
  typeof(string),
  FieldAttributes.Private);
```

注意TypeBuilder.DefineField()方法如何提供到FieldBuilder类型的访问。TypeBuilder类也提供了到其他构造类型的访问方法。例如，DefineConstructor()返回一个ConstructorBuilder，DefineProperty()返回一个PropertyBuilder，诸如此类。

18.10.7 产生构造函数

前面提到过，TypeBuilder.DefineConstructor()方法可以用来为当前类型定义构造函数。不过，当需要实现HelloClass的构造函数时，还需要注入原始CIL代码到构造函数内来接收传入的参数并赋值到内部的私有字符串。要获得ILGenerator类型，需要调用引用的“builder”类型（本例中是ConstructorBuilder类型）的GetILGenerator()方法。

ILGenerator类的Emit()方法是负责把CIL放入成员实现的实体。Emit()自身频繁使用OpCodes类类型，这个类使用只读字段公开CIL的操作码集。例如，OpCodes.Ret表示一个方法调用的返回；OpCodes.Stfld给成员变量赋值；OpCodes.Call可以调用一个指定的方法（在本例中是基类构造函数）。

参考下面构造函数的逻辑：

```
// 创建一个自定义的构造函数，它只有一个System.String参数
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
  helloWorldClass.DefineConstructor(MethodAttributes.Public,
  CallingConventions.Standard, constructorArgs);

// 产生必要的CIL代码到这个构造函数
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
ConstructorInfo superConstructor = objectClass.GetConstructor(new Type[0]);
constructorIL.Emit(OpCodes.Call, superConstructor);  // 调用基类的构造函数

// 加载对象的this指针到栈上
constructorIL.Emit(OpCodes.Ldarg_0);

// 加载输入参数到虚拟栈上并存储在msgField中
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField);  // 赋值msgField
constructorIL.Emit(OpCodes.Ret);               // 返回
```

现在读者已经发现，只要给类型定义了自定义的构造函数，默认的构造函数就被悄然删除了。要想重新定义一个没有参数的构造函数，只要调用TypeBuilder类型的DefineDefaultConstructor()方法：

```
// 重新插入默认构造函数
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);
```

这个调用产生标准的CIL代码，用来定义默认的构造函数：

```
.method public hidebysig specialname rtspecialname
  instance void  .ctor() cil managed
{
  .maxstack  1
  ldarg.0
  call instance void [mscorlib]System.Object::.ctor()
  ret
}
```

18.10.8　产生SayHello()方法

最后，让我们看看如何产生SayHello()方法。首先需要做的是从helloWorldClass变量获得MethodBuilder类型。一旦得到这个类型，就可以定义方法和获得ILGenerator来注入CIL指令：

```
// 创建SayHello()方法
MethodBuilder sayHiMethod =
  helloWorldClass.DefineMethod("SayHello",
  MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();

// 输出一行到控制台
methodIL.EmitWriteLine("Hello there!");
methodIL.Emit(OpCodes.Ret);
```

这里创建了一个没有参数、什么也不返回（用DefineMethod()调用中包含的空项表示）的公共方

法（MethodAttributes.Public）。另外请关注下EmitWriteLine()调用。这个ILGenerator类的辅助成员自动输出一行到标准输出中。

18.10.9 使用动态产生的程序集

现在已经构造好了创建和保存程序集的逻辑，只需要一个类来触发这个逻辑。假设你的项目定义了第二个类AsmReader。Main()函数的逻辑通过Thread.GetDomain()方法得到当前的AppDomain，AppDomain被用来承载动态创建的程序集。一旦有了引用，就可以调用CreateMyAsm()方法。

为了让我们的程序更加有趣，在调用的CreateMyAsm()方法返回后，可以练习一下延迟绑定（见第15章）来加载新创建的程序集到内存中并且和HelloWorld类的成员交互。按如下所示修改Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Amazing Dynamic Assembly Builder App *****");
  // 得到当前线程的应用程序域
  AppDomain curAppDomain = Thread.GetDomain();

  // 使用辅助函数f(x)创建动态程序集
  CreateMyAsm(curAppDomain);
  Console.WriteLine("-> Finished creating MyAssembly.dll.");

  // 加载新的程序集
  Console.WriteLine("-> Loading MyAssembly.dll from file.");
  Assembly a = Assembly.Load("MyAssembly");

  // 得到HelloWorld类型
  Type hello = a.GetType("MyAssembly.HelloWorld");

  // 创建HelloWorld对象并调用正确的构造函数
  Console.Write("-> Enter message to pass HelloWorld class: ");
  string msg = Console.ReadLine();
  object[] ctorArgs = new object[1];
  ctorArgs[0] = msg;
  object obj = Activator.CreateInstance(hello, ctorArgs);

  // 调用SayHello()并且显示返回的字符串
  Console.WriteLine("-> Calling SayHello() via late binding.");
  MethodInfo mi = hello.GetMethod("SayHello");
  mi.Invoke(obj, null);

  // 触发GetMsg()
  mi = hello.GetMethod("GetMsg");
  Console.WriteLine(mi.Invoke(obj, null));
}
```

事实上，至此你已经创建了一个可以在运行时创建和执行程序集的.NET程序集。这个例子结束了对CIL和动态程序集作用的学习。希望这章能够加深读者对.NET类型系统以及CIL语法和语义的理解。

源代码 DynamicAsmBuilder项目的源代码位于Chapter 18子目录下。

18.11 小结

在这一章里，我们学习了CIL的语法和语义。与C#这样的高级托管语言不同，CIL不仅定义了一系列关键字，而且提供了指令（用来定义程序集的结构及其类型）、特性（更进一步定义了一个给定指令）以及操作码（用来实现类型成员）。

另外，还介绍了CIL的编程工具（ilasm.exe、sharpDevelop和peverify.exe），并且学习了如何用正反向工程通过新的CIL代码修改.NET程序集的内容。在此之后，我们花时间学习了如何建立当前的（和被引用的）程序集、命名空间、类型和成员，然后介绍了一个简单的示例，使用CIL、命令行工具和一些脑细胞构建了一个.NET代码库和可执行文件。

最后介绍了创建动态程序集的过程。使用System.Reflection.Emit命名空间，可以在运行时在内存中定义.NET程序集。正如你所看到的，使用这个特殊的API要求你详细了解CIL代码的语义。对于大多数.NET程序而言，构建动态程序集并不是常见的任务，但对那些需要构建支持工具和其他编程实用工具的人来说，是十分有用的。

Part 6

第六部分

.NET 基础类库

本部分内容

第19章 多线程、并行和异步编程

没人喜欢运行缓慢迟钝的应用程序，也没人喜欢打开的某个新任务影响程序其他部分的响应速度。在.NET发布之前，构建具备执行多任务能力的应用程序需要编写极其复杂的使用Windows线程API的C++代码。幸好.NET平台提供了许多方法，可以很轻松地构建在特殊的执行路径执行复杂操作的软件。

首先，我们定义"多线程应用程序"的总体性质，然后回顾.NET的委托类型，并理解它对异步方法调用的内在支持。你将看到，这项技术允许用户自动地在次线程中调用某个方法，而不需要手工创建或配置线程。

接着，将研究原始的线程命名空间，即从.NET 1.0起就发布了的System.Threading。这里的许多类型（比如Thread、ThreadStart等）可以显式地创建更多执行线程并同步共享资源。它们可以确保多线程以非易失（nonvolatile）的方式共享数据。

本章剩下的部分将介绍近期的三项可供.NET开发者构建多线程软件的技术，它们是任务并行库（TPL）、并行LINQ（PLINQ）和C#中新的内置异步关键字（async和await）。你将看到，这些特性可以显著地简化构建快速响应的多线程应用程序的过程。

19.1 进程、应用程序域、上下文及线程之间的关系

在第17章中，线程被定义为可执行应用程序中的基本执行单元。虽然许多.NET程序在单线程模式下运行得很好，但程序集的主线程（在Main()方法执行时由CLR产生的）可能随时创建次线程，来执行一些额外的工作单元。通过创建这些新增的线程，能构建出响应更快（但在单核机器上不一定执行更快）的应用程序。

System.Threading随.NET 1.0发布，它提供了一些创建多线程应用程序的途径。Thread类是核心，它代表了某个给定的线程。若想要通过编程得到对当前（正在执行某段代码的）线程的引用，只需要调用静态属性Thread.CurrentThread，如下所示：

```
static void ExtractExecutingThread()
{
  // 得到正在执行这个方法的线程
  Thread currThread = Thread.CurrentThread;
}
```

在.NET平台下，应用程序域和线程之间并不是一一对应的。事实上，在任何时间，一个应用程序域内都可能有多个线程。而且，一个特定的线程在它的生命周期内并不一定被限定在一个应用程

序域中。Windows OS线程调度程序和.NET CLR会根据需要让线程能够自由地跨越应用程序域的边界。

虽然活动的线程能够跨越多个应用程序域边界，但是在任何一个时间点上，一个线程只能运行在一个应用程序域中（也就是说，一个线程同时在多个应用程序域上执行任务是不可能的）。当希望访问（正在承载当前线程的）应用程序域时，请调用静态方法Thread.GetDomain()，如下所示：

```
static void ExtractAppDomainHostingThread()
{
  // 获取正在承载当前线程的应用程序域
  AppDomain ad = Thread.GetDomain();
}
```

在任何特定的时刻，一个线程也可以移动到一个特定的上下文中，并且它可以由CLR重新部署在一个新的上下文中。要获得正在执行中的线程的当前上下文，请调用静态属性Thread.CurrentContext（它返回System.Runtime.Remoting.Contexts.Context对象），如下所示：

```
static void ExtractCurrentThreadContext()
{
  // 获取当前操作线程所处的上下文
  Context ctx = Thread.CurrentContext;
}
```

此外，CLR是控制线程移入/移出应用程序域和上下文的实体。作为.NET开发人员，你通常无须知道一个线程在哪里结束（或者准确地说，它是在什么时候被放置在新的边界上的），但是，至少应当知道获得底层原语（underlying primitive）的不同方法。

19.1.1 并发问题

多线程编程的“乐趣”（苦中作乐）之一是：几乎无法控制底层操作系统和CLR对线程的调度。举例来说，如果精心编写一段创建一个新线程的代码，你不能保证这个线程被立即执行。更准确地说，这段代码仅仅通知操作系统或CLR尽快地执行这个线程（通常是线程调度程序给这个线程分配时间）。

此外，既然通过CLR线程可以在应用程序域和上下文边界之间移动，就必须留心应用程序中的线程不稳定（thread-volatile）操作（如在多线程访问的情况下）和原子型（atomic）操作。要知道，线程不稳定操作是很危险的。

举例来说，假设有一个线程正在调用某个特定对象的一个方法，为了让另一个线程也访问同一对象的同一方法，线程调度程序将发出指令挂起第一个线程。

而此时，如果前一个线程没有全部完成当前的操作，那么后来的线程可能看到对象处于被部分修改状态。这样它所读到的数据基本上是虚假的，而这会使应用程序发生非常奇怪的（并且是非常难以发现的）bug，而且这些bug都难以重现和调试。

另一方面，原子型操作在多线程环境下总是（线程）安全的。可令人沮丧的是，.NET基础类库中只有很少的操作能保证原子型。甚至将一个值赋给一个成员变量的操作也不是原子型的。因此，如果在.NET Framework 4.5 SDK中没有明确指明一个操作是原子型的，那么请先假定它是线程不稳定的并小心对待。

19.1.2　线程同步的作用

因此很明显，多线程程序本身是相当不稳定的，因为在同一时间，多个线程都能（或多或少地）运行共享的功能块。为了保护应用程序的资源不被破坏，.NET开发者必须使用线程的各种原语（比如lock、monitor和[Synchronization]特性或语言关键字支持）来控制线程对它们的访问。

虽然在.NET平台下，构建一个健壮的多线程程序的困难并没有完全克服，但这个复杂过程还是被大大简化了。使用System.Threading命名空间中定义的类型、.NET 4.0或更高版本的TPL（Task Parallel Library，任务并行库）以及.NET 4.5中C#的async和await语言关键字，可以比较省心地操作多线程。

在深入介绍System.Threading命名空间、TPL和C# async和await关键字之前，我们先来学习.NET委托如何以异步的方式调用方法。尽管.NET 4.5中的C# async和await关键字肯定比异步委托更加简单，但了解如何以这种方式与代码交互仍然是十分重要的（相信我，在生产代码中有相当一部分使用了异步委托）。

19.2　.NET委托的简短回顾

回想一下，.NET委托是一个类型安全的、面向对象的函数指针。当定义一个.NET委托类型时，作为响应，C#编译器将创建一个派生自System.MulticastDelegate的密封类（而System.MulticastDelegate又派生自System.Delegate）。这些基类为每一个委托提供了维护方法地址列表的能力，这些方法可以在以后被调用。让我们来看一看下面的BinaryOp委托（在第10章中被首次定义）。

```
// C#委托类型
public delegate int BinaryOp(int x, int y);
```

从BinaryOp的定义来看，它能够指向任何一个拥有两个整型参数、返回一个整数的方法。BinaryOp被编译后，其所属程序集将包含一个根据委托声明动态生成的类的定义。在BinaryOp的例子中，该类看起来差不多是下面的样子（以伪代码的方式）：

```
public sealed class BinaryOp : System.MulticastDelegate
{
  public BinaryOp(object target, uint functionAddress);
  public int Invoke(int x, int y);
  public IAsyncResult BeginInvoke(int x, int y,
    AsyncCallback cb, object state);
  public int EndInvoke(IAsyncResult result);
}
```

回想一下，生成的Invoke()方法用来调用被代理对象以同步方式维护的方法。因此，调用委托的线程（比如应用程序的主线程）将会一直等待，直到委托调用完成。此外，在C#中，Invoke()方法并不会直接在代码中被调用，而是在使用“正常的”方法调用语法时在幕后被触发的。

考虑下面的控制台程序（SyncDelegateReview），它在同步（又称阻塞）模式下调用了静态方法Add()（请一定在C#代码文件中导入System.Threading命名空间，因为你将调用Thread.Sleep()方法）：

```
namespace SyncDelegateReview
{
  public delegate int BinaryOp(int x, int y);
```

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Synch Delegate Review *****");

    // 输出正在执行中的线程ID
    Console.WriteLine("Main() invoked on thread {0}.",
      Thread.CurrentThread.ManagedThreadId);

    // 在同步模式下调用Add()
    BinaryOp b = new BinaryOp(Add);

    // 也能写b.Invoke(10,10);
    int answer = b(10, 10);

    // 直到Add()方法完成后，这行代码才会执行
    Console.WriteLine("Doing more work in Main()!");
    Console.WriteLine("10 + 10 is {0}.", answer);
    Console.ReadLine();
  }

  static int Add(int x, int y)
  {
    // 输出正在执行中的线程ID
    Console.WriteLine("Add() invoked on thread {0}.",
      Thread.CurrentThread.ManagedThreadId);

    // 暂停一下，模拟一个耗时的操作
    Thread.Sleep(5000);
    return x + y;
  }
 }
}
```

在Add()方法中，为了模拟一个耗时很多的操作，我们调用了静态方法Thread.Sleep()来使当前线程挂起（差不多）五秒钟。因为你是在同步模式下调用Add()方法，Main()方法将会等到Add()方法完成才输出操作的结果。

接下来，注意一下Main()方法，它（通过Thread.CurrentThread）获得对当前线程的访问，然后通过ManagedThreadId属性输出线程的ID来。这一逻辑也会出现在静态方法Add()中。正如所料，由于程序中所有的任务都被主线程执行，控制台中显示的将是相同的ID值：

```
***** Synch Delegate Review *****
Main() invoked on thread 1.
Add() invoked on thread 1.
Doing more work in Main()!
10 + 10 is 20.

Press any key to continue . . .
```

当运行这个程序的时候，请注意在Main()中的Console.WriteLine()逻辑执行之前，有5秒的延迟发生。虽然许多方法可以被同步调用，且没有负面影响，但如果需要的话，.NET委托能够以异步方式调用方法。

源代码　SyncDelegateReview项目的源代码位于Chapter 19子目录下。

19.3　委托的异步性

如果刚接触多线程，读者可能想知道究竟什么是异步方法调用。不用想就知道，有些程序操作会花费较长时间。前面的Add()虽然纯粹是示例，但是设想一下这样的情况：一个单线程程序调用一个远程对象的方法，调用一个执行耗时数据库查询的方法，下载一个大文档，或向一个外部文件写500行的文字。在执行这些操作时，应用程序会显得挂起很长时间。在任务完成之前，这个程序的其他部分（比如菜单激活、工具条的单击或者控制台的输出）都会挂起（这会让用户不耐烦）。

因此问题是，如何使委托在单独的线程上调用方法，以便模拟多个“同时”运行的任务？好消息是每一个.NET委托类型自动配备了这项能力，更好的消息是不需要深入研究System.Threading命名空间的细节就能实现（虽然这些实体可以自然地联合工作）。

19.3.1　BeginInvoke()和EndInvoke()方法

C#编译器处理delegate关键字的时候，其动态生成的类定义了两个方法——BeginInvoke()和EndInvoke()。基于我们对BinaryOp委托的定义，这些方法的原型如下：

```
public sealed class BinaryOp : System.MulticastDelegate
{
...
  // 用于异步调用方法
  public IAsyncResult BeginInvoke(int x, int y,
    AsyncCallback cb, object state);

  // 用于获取被调用方法的返回值
  public int EndInvoke(IAsyncResult result);
}
```

传入BeginInvoke()的参数的最初集合必须符合C#委托约定（对于BinaryOp，就是两个整型）。最后两个参数必须是System.AsyncCallback和System.Object。稍后我们将研究这两个参数的作用，目前，暂时把null值赋给它们。根据BinaryOp的返回类型，EndInvoke()的返回值是整型，而这个方法的唯一参数总是IAsyncResult类型。

19.3.2　System.IAsyncResult接口

另外，BeginInvoke()返回的对象实现了IAsyncResult接口，而EndInvoke()需要一个IAsyncResult兼容（即实现了IAsyncResult接口的类型）类型作为它唯一的参数。由BeginInvoke()返回的IAsyncResult兼容对象主要是一种耦合机制，它允许调用的线程在稍后通过EndInvoke()获取异步方法调用的结果。IAsyncResult接口（在System命名空间中定义）的定义如下：

```
public interface IAsyncResult
{
  object AsyncState { get; }
  WaitHandle AsyncWaitHandle { get; }
```

```
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
  }
```

在这个最简单的例子中，不需要直接调用IAsyncResult的成员。需要做的全部事情就是缓存自BeginInvoke()返回的IAsyncResult兼容类型，并在准备获取方法调用的结果时，把它传给EndInvoke()。可以看到，当想更深入到获取方法返回值的过程时，可以调用IAsyncResult兼容的成员。

说明 如果异步调用一个无返回值的方法，仅仅调用BeginInvoke()就可以了。在这种情况下，我们不需要缓存IAsyncResult兼容对象，也不需要首先调用EndInvoke()（因为没有收到返回值）。

19.4 异步调用方法

为了通知BinaryOp委托异步调用Add()，需要修改上面项目中的逻辑（你可以在已有的项目中添加代码，但在下载的代码包中有一个新建的控制台应用程序，名为AsyncDelegate）。需要像下面这样修改前面的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Async Delegate Invocation *****");

  // 输出正在执行中的线程的ID
  Console.WriteLine("Main() invoked on thread {0}.",
    Thread.CurrentThread.ManagedThreadId);

  // 在次线程中调用Add()
  BinaryOp b = new BinaryOp(Add);
  IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

  // 在主线程中做其他事情
  Console.WriteLine("Doing more work in Main()!");

  // 当执行完后获取Add()方法的结果
  int answer = b.EndInvoke(iftAR);
  Console.WriteLine("10 + 10 is {0}.", answer);
  Console.ReadLine();
}
```

如果运行这个程序，我们将发现两个不同的ID值，这说明事实上在当前应用程序域中有两个线程正在运行

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
10 + 10 is 20.
```

除了显示的ID值不同，只要一运行程序，消息“Doing more work in Main()!”立即就显示出来了，而次线程正在忙于处理业务。

19.4.1　同步调用线程

如果仔细思考一下Main()的实现，可能意识到BeginInvoke()和EndInvoke()之间的时间间隔明显小于5秒钟，因此，在“Doing more work in Main()!”被输出到控制台后，主线程将会被阻塞，并一直等到次线程完成才能获得Add()方法的结果。这样效率似乎不太高，我们需要做另一个同步调用：

```
static void Main(string[] args)
{
...
  BinaryOp b = new BinaryOp(Add);

  // 一旦下一条语句被处理，调用线程在BeginInvoke()完成之前就被阻塞了
  IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

  // 调用的时间远远小于5秒钟
  Console.WriteLine("Doing more work in Main()!");

  // 现在再次等待其他线程完成
  int answer = b.EndInvoke(iftAR);
...
}
```

很明显，在不同环境下，如果调用线程（这里指主线程）有被阻塞的可能，那么异步委托就毫无优势可言。为了让调用线程能够发现异步调用是否完成，IAsyncResult接口提供了IsCompleted属性。使用这个成员，调用线程在调用EndInvoke()之前，便能够判断异步调用是否真正完成。

如果方法没有完成，IsCompleted返回false，这时调用线程可以自由地做其他事情。如果IsCompleted返回true，调用线程便可能以最小的阻塞代价获得返回结果。仔细思考一下下面Main()方法中的改动：

```
static void Main(string[] args)
{
...
  BinaryOp b = new BinaryOp(Add);
  IAsyncResult iftAR = b.BeginInvoke(10, 10, null, null);

  // 直到Add()方法完成，消息才会显示出来
  while(!iftAR.IsCompleted)
  {
    Console.WriteLine("Doing more work in Main()!");
    Thread.Sleep(1000);
  }
  // 现在我们知道Add()完成了
  int answer = b.EndInvoke(iftAR);
...
}
```

这里，在次线程完成之前，循环将不断地执行Console.WriteLine()语句。一旦次线程完成，我们便能够确信Add()方法真正完成了，从而获得Add()方法的返回值。然而，调用Thread.Sleep(1000)对于这个应用程序的正常工作来说不是必需的，但是它可以强制主线程在每次迭代之后等待大约1秒，也可以防止相同的消息输出几百次。以下是输出结果（根据机器的速度和线程启动的时间，输出结果会稍有不同）：

```
***** Async Delegate Invocation *****
Main() invoked on thread 1.
Doing more work in Main()!
Add() invoked on thread 3.
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
Doing more work in Main()!
10 + 10 is 20.
```


除IsCompleted属性之外，IAsyncResult接口提供了AsyncWaitHandle属性以实现更加灵活的等待逻辑。这个属性返回一个WaitHandle类型的实例，该实例公开了一个名为WaitOne()的方法。使用WaitHandle.WaitOne()的好处是可以指定最长等待时间。如果超时，WaitOne()返回false。考虑while循环中的如下变化，它没有调用Thread.Sleep()：

```
while (!iftAR.AsyncWaitHandle.WaitOne(1000, true))
{
  Console.WriteLine("Doing more work in Main()!");
}
```

虽然IAsyncResult的这些属性提供了同步调用线程的方式，但是这不是最高效的方式。总地来说，IsCompleted属性就像一个令人讨厌的经理（或者同学），他总是不断地问："你完成了吗？"谢天谢地，委托提供了另外的（也是更有效的）技术来获取异步调用的结果。

源代码 AsyncDelegate项目的源代码位于Chapter 19子目录下。

19.4.2 AsyncCallback委托的作用

不通过轮询一个委托来确定异步调用方法执行是否结束，而是在任务完成时由次线程主动通知调用线程的方式，这样可能更好。如果想要实现这种方式，需要在调用BeginInvoke()时提供一个System.AsyncCallback委托的实例作为参数，这个参数的默认值是null。只要提供了AsyncCallback对象，当异步调用完成的时候，委托便会自动调用（AsyncCallback对象）指定的方法。

说明 回调方法将在次线程而不是主线程中调用。这在图形用户界面（WPF或Windows Forms）中使用线程时，具有重要的意义。因为控件都是与线程紧密相关的，只能由创建它们的线程进行操作。你将在本章稍后介绍任务并行库（TPL）和.NET 4.5中新的C# async和await关键字时看到一些在GUI中使用线程的示例。

和所有委托一样，AsyncCallback委托仅仅能够调用那些符合特定模式的方法，这些方法只有一个参数IAsyncResult，而且没有返回值。

```
// AsynCallback的目标必须和下面的模式相匹配
void MyAsyncCallbackMethod(IAsyncResult itfAR)
```

假定有另外一个控制台应用程序（AsyncCallbackDelegate）使用了BinaryOp委托。但这次不需要轮询委托以判断Add()方法是否执行完毕，而是定义一个名为AddComplete()的静态方法，用它来接受异步调用完成的通知。同样，该示例使用类级别的静态bool字段，它确保Main()中的主线程持续运行一个任务，直到次线程完成。

说明　严格地说，这样使用Boolean变量不是线程安全的，因为有两个不同的线程访问它的值。这在当前示例中没有问题，但一般来说，你必须确保由多个线程共享的数据是锁定的。本章稍后将介绍这种做法。

```
namespace AsyncCallbackDelegate
{
  public delegate int BinaryOp(int x, int y);

  class Program
  {
    private static bool isDone = false;

    static void Main(string[] args)
    {
      Console.WriteLine("*****  AsyncCallbackDelegate Example *****");
      Console.WriteLine("Main() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);

      BinaryOp b = new BinaryOp(Add);
      IAsyncResult iftAR = b.BeginInvoke(10, 10,
        new AsyncCallback(AddComplete), null);

      // 这里可以做其他的事情
      while (!isDone)
      {
        Thread.Sleep(1000);
        Console.WriteLine("Working....");
      }
      Console.ReadLine();
    }

    static int Add(int x, int y)
    {
      Console.WriteLine("Add() invoked on thread {0}.",
      Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(5000);
      return x + y;
    }

    static void AddComplete(IAsyncResult itfAR)
    {
      Console.WriteLine("AddComplete() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
      Console.WriteLine("Your addition is complete");
      isDone = true;
    }
  }
}
```

当Add()执行完成的时候，AsyncCallback委托将调用静态的AddComplete()方法。如果运行上面这个程序，可以确定次线程是回调AddComplete()的线程：

```
***** AsyncCallbackDelegate Example *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
```

和本章的其他示例一样，输出结果会稍有不同。事实上，你很可能看到在其他内容输出完毕后还有"Working..."出现。这是Main()中1秒延迟的副产品。

19.4.3 AsyncResult类的作用

当前，AddComplete()方法并没有输出操作（两个数的和）的实际结果。这样做的原因是：AsyncCallback委托的目标（即本例的AddComplete()）无法访问在Main()中创建的BinaryOp委托。因此，不能在AddComplete()中调用EndInvoke()。

虽然可以把BinaryOp定义成静态的，以便两个方法都能访问相同的对象，但更好的解决方案是采用IAsyncResult输入参数。

IAsyncResult输入参数被传入了AsyncCallback委托目标中，它实际上是定义在System.Runtime.Remoting.Messaging命名空间下的AsyncResult类（注意：没有前缀"I"）的一个实例。该类的只读属性AsyncDelegate返回了别处创建的原始异步委托的引用。

因此，如果想获取对分配在Main()中的BinaryOp委托对象的引用，只需把由AsyncDelegate属性返回的System.Object类型转换成BinaryOp类型就可以了。这时，终于可以触发EndInvoke()方法了：

```
// 不要忘记导入System.Runtime.Remoting.Messaging
static void AddComplete(IAsyncResult itfAR)
{
  Console.WriteLine("AddComplete() invoked on thread {0}.",
    Thread.CurrentThread.ManagedThreadId);
  Console.WriteLine("Your addition is complete");

  // 现在得到结果
  AsyncResult ar = (AsyncResult)itfAR;
  BinaryOp b = (BinaryOp)ar.AsyncDelegate;
  Console.WriteLine("10 + 10 is {0}.", b.EndInvoke(itfAR));
  isDone = true;
}
```

19.4.4 传递和接收自定义状态数据

异步委托的最后一个需要关注的地方是BeginInvoke()方法的最后一个参数（默认为null）。该参

数允许从主线程传递额外的状态信息给回调方法。因为这个参数类型是System.Object，所以可以传入任何回调方法所希望的类型的数据。在下面的示例中，主线程将向AddComplete()方法传入一个自定义的文本消息：

```
static void Main(string[] args)
{
...
  IAsyncResult iftAR = b.BeginInvoke(10, 10,
    new AsyncCallback(AddComplete),
    "Main() thanks you for adding these numbers.");
...
}
```

为了在AddComplete()中获得数据，使用传入IAsyncResult参数的AsyncState属性。注意，这里会要求显式强制转换，因此，主线程和次线程必须认同由AsyncState返回的实际类型。

```
static void AddComplete(IAsyncResult itfAR)
{
...
  // 获取消息对象，并转换成string
  string msg = (string)itfAR.AsyncState;
  Console.WriteLine(msg);
  isDone = true;
}
```

下面显示了当前程序的输出结果：

```
*****  AsyncCallbackDelegate Example *****
Main() invoked on thread 1.
Add() invoked on thread 3.
Working....
Working....
Working....
Working....
Working....
AddComplete() invoked on thread 3.
Your addition is complete
10 + 10 is 20.
Main() thanks you for adding these numbers.
```

我们已经理解了如何使用.NET委托来自动创建次线程以处理异步方法调用，接下来，讨论如何用System.Threading命名空间与线程进行交互。记住，它是从.NET 1.0就存在的最早的线程API。

源代码 AsyncCallbackDelegate项目的源代码位于Chapter 19子目录下。

19.5 System.Threading 命名空间

在.NET平台下，System.Threading命名空间提供了许多类型用来构建多线程应用程序。除了和特殊的CLR线程进行交互的类型外，这个命名空间还定义了许多其他类型，这些类型允许访问CLR维护

的线程池、一个简单（无界面的）Timer类，以及大量用来同步访问共享资源的类型。表19-1列出了该命名空间中的重要成员。（务必参考.NET Framework 4.5 SDK文档以得到细节描述。）

表19-1 System.Threading命名空间中的部分类型

类型	作用
Interlocked	为被多个线程共享访问的类型提供原子操作
Monitor	使用锁定和等待信号来同步线程对象。C#的lock关键字在后台使用的就是Monitor对象
Mutex	互斥体，可用于应用程序域边界之间的同步
ParameterizedThreadStart	委托，它允许线程调用包含任意多个参数的方法
Semaphore	用于限制对一个资源或一类资源的并发访问的线程数量
Thread	代表CLR中执行的线程。使用这个类型，能够在初始的应用程序域中创建额外的线程
ThreadPool	用于和一个进程中的（由CLR维护的）线程池交互
ThreadPriority	代表了线程调度的优先级别（Highest、Normal等）
ThreadStart	该委托用于定义一个线程所调用的方法。和ParameterizedThreadStart委托不同，这个方法的目标必须符合一种固定的原型
ThreadState	代表线程处于的状态（Running、Aborted等）
Timer	提供以指定的时间间隔执行方法的机制
TimerCallback	该委托类型应与Timer类型一起使用

19.6 System.Threading.Thread 类

System.Threading命名空间中最基本的类型是Thread。它是一个面向对象的包装器，包装特定应用程序域中的某个执行单元。Thread类型中定义了许多方法（包括静态的和共享的），使用这些方法能够在当前应用程序域中创建、挂起、停止和销毁线程。表19-2列出了Thread类的核心静态成员。

表19-2 Thread类型的主要静态成员

静态成员	作用
CurrentContext	只读属性，返回当前线程的上下文
CurrentThread	只读属性，返回当前线程的引用
GetDomain()和GetDomainID()	返回当前应用程序域的引用或当前线程正在运行的域的ID
Sleep()	将当前线程挂起指定的时间

Thread类也支持部分实例级的成员，表19-3中列出了其中的一部分。

表19-3 线程类型的主要实例级成员

实例级成员	作 用
IsAlive	返回布尔值，指示线程是否开始了
IsBackground	获取或设置一个值，指示线程是否为后台线程（后面有详细说明）
Name	给线程指定的友好的名字
Priority	获取或设置线程的调度优先级。它是ThreadPriority枚举中的值之一
ThreadState	获取当前线程的状态。它是ThreadState枚举中的值之一
Abort()	通知CLR尽快终止本线程
Interrupt()	中断当前线程，唤醒处于等待中的线程
Join()	阻塞调用线程，直到某个（调用Join()的）线程终止为止
Resume()	使已挂起的线程继续执行
Start()	通知CLR尽快执行本线程
Suspend()	挂起当前线程，如果线程已挂起，调用Suspend()则不起作用

说明 终止或挂起活动线程通常不是一个好主意。这样可能（当然，可能性很小）会导致线程在受到扰乱或终止时“泄露”工作量。

19.6.1 获得当前执行线程的统计信息

回想一下，可执行程序集的入口点（即Main()方法）是运行在主线程上的。为举例说明Thread类型的基本用法，假定有一个新的名为ThreadStats的控制台程序。如读者所知，静态的Thread.CurrentThread属性找到表示当前正在执行线程的Thread对象。一旦获得了当前线程，便能够输出各种统计信息。如下所示：

```
// 确认引入System.Threading命名空间
static void Main(string[] args)
{
  Console.WriteLine("***** Primary Thread stats *****\n");

  // 获取当前线程的名字
  Thread primaryThread = Thread.CurrentThread;
  primaryThread.Name = "ThePrimaryThread";

  // 显示承载的应用程序域和上下文的详细信息
  Console.WriteLine("Name of current AppDomain: {0}",
    Thread.GetDomain().FriendlyName);
  Console.WriteLine("ID of current Context: {0}",
    Thread.CurrentContext.ContextID);

  // 输出线程的一些信息
  Console.WriteLine("Thread Name: {0}",
    primaryThread.Name);
  Console.WriteLine("Has thread started?: {0}",
    primaryThread.IsAlive);
  Console.WriteLine("Priority Level: {0}",
```

```
    primaryThread.Priority);
  Console.WriteLine("Thread State: {0}",
    primaryThread.ThreadState);
  Console.ReadLine();
}
```

下面显示了这个程序的输出结果：

```
***** Primary Thread stats *****

Name of current AppDomain: ThreadStats.exe
ID of current Context: 0
Thread Name: ThePrimaryThread
Has thread started?: True
Priority Level: Normal
Thread State: Running
```

19.6.2 Name属性

虽然上述代码或多或少有自描述的性质，但是请注意Thread类支持Name属性。如果没有设置这个值的话，Name将返回一个空的字符串。然而，一旦为一个给定的线程对象指定一个友好的字符串名字，在调试的时候就简单多了。如果使用Visual Studio，可以在调试的时候访问Threads窗口（在菜单上选择Debug→Windows→Threads）。如图19-1所示，能够很容易地找到想调试的线程。

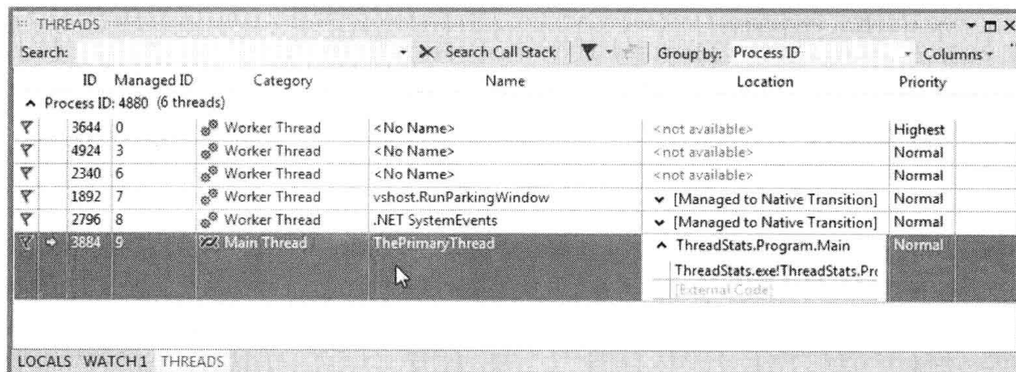


图19-1 在Visual Studio中调试线程

19.6.3 Priority属性

接下来，注意Thread类型定义了一个名为Priority的属性，默认情况下，所有线程的优先级都处于Normal级别。但是，在线程生命周期的任何时候，都可以使用ThreadPriority属性修改线程的优先级，且修改的值必须是System.Threading.ThreadPriority枚举中的一个，如下所示：

```
public enum ThreadPriority
{
```

```
    Lowest,
    BelowNormal,
    Normal,  // 默认值
    AboveNormal,
    Highest
  }
```

如果给线程的优先级指定一个非默认值（默认值为ThreadPriority.Normal），应当知道这并不能控制线程调度器切换线程的过程。实际上，一个线程的优先级仅仅是把线程活动的重要程度提供给CLR。因此，一个带有ThreadPriority.Highest优先级的线程并不一定保证能得到最高的优先级。

此外，如果线程调度器被某个任务（比如同步对象、切换线程或线程移动）抢占了，那么线程的优先级别很有可能因此会被修改。这个时候，CLR将会读取这些值，并指示线程调度器如何最好地分配时间片。总地来说，有着相同优先调度级别的线程应当得到相同数量的时间来执行任务。

大多数情况下，我们几乎不需要直接来改变线程的优先级别。理论上，提高一些线程的优先级别会阻止那些低优先级别的线程执行任务（所以需要小心使用）。

源代码　ThreadStats项目的源代码位于Chapter 19子目录下。

19.7　手工创建次线程

当希望以编程方式创建次线程以分担一些工作单元时，在使用System.Threading命名空间下的类型时可以遵从下面预定的步骤。

(1) 创建一个方法作为新线程的入口点。

(2) 创建一个ParameterizedThreadStart（或者ThreadStart）委托，并把在上一步所定义方法的地址传给委托的构造函数。

(3) 创建一个Thread对象，并把ParameterizedThreadStart或ThreadStart委托作为构造函数的参数。

(4) 建立任意初始化线程的特性（名称、优先级等）。

(5) 调用Thread.Start()方法。在第(2)个步骤中建立的委托所指向的方法将在线程中尽快开始执行。

按步骤(2)的规定，可以使用两种不同的委托类型指向调用将在次线程中执行的方法。ThreadStart委托指向一个没有参数、无返回值的方法。这个委托在一个方法被设计用来仅仅在后台运行、而没有更多的交互时非常有用。

很明显，ThreadStart的局限是用户无法给过程传递参数。但是ParameterizedThreadStart委托类型允许包含一个System.Object类型的参数。由于任何对象都源于System.Object，所以可以通过一个自定义的类或结构来传递任意数量的参数[①]。但需要注意的是，ParameterizedThreadStart委托仅仅指向无返回值的方法。

19.7.1　使用ThreadStart委托

举例说明构建多线程程序的过程（同时也是展示这样做的好处），假定有一个控制台程序

① 即可以自定义一个类/结构，这个类/结构包含了所有的参数。——译者注

（SimpleMultiThreadApp），它能够让用户选择是采用单线程还是两个分离的线程来执行任务。

假定已经引入了System.Threading命名空间，首先需要定义一个方法，它（可能）在次线程执行任务。关注多线程程序的构建机制，这个方法仅仅在控制台窗口上按顺序循环输出一些数字，每一次循环大概暂停两秒钟。下面是Printer类完整的定义：

```
public class Printer
{
  public void PrintNumbers()
  {
    // 显示Thread信息
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);

    // 输出数字
    Console.Write("Your numbers: ");
    for(int i = 0; i < 10; i++)
    {
      Console.Write("{0}, ", i);
      Thread.Sleep(2000);
    }
    Console.WriteLine();
  }
}
```

在Main()方法中，首先要提示用户，让他来确定是用一个还是两个线程来执行任务。如果用户需要一个单独线程，那么在主线程上调用PrintNumbers()方法就可以了。但是，如果用户指定需要两个线程，那么创建一个指向PrintNumbers()方法的ThreadStart委托，接着把这个委托对象传给一个新创建的Thread对象的构造函数，并且调用这个Thread对象的Start()方法以通知CLR：线程已经准备好执行了。

首先，添加System.Windows.Forms.dll程序集引用和System.Windows.Forms命名空间，并且在Main()中使用MessageBox.Show()显示一条消息（一旦运行程序，就能看到它执行到什么地方）。下面是完整的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Amazing Thread App *****\n");
  Console.Write("Do you want [1] or [2] threads? ");
  string threadCount = Console.ReadLine();

  // 命名当前线程
  Thread primaryThread = Thread.CurrentThread;
  primaryThread.Name = "Primary";

  // 显示线程的信息
  Console.WriteLine("-> {0} is executing Main()",
  Thread.CurrentThread.Name);

  // 创建执行任务的类
  Printer p = new Printer();

  switch(threadCount)
  {
    case  "2":
      // 设置线程
```

```
        Thread backgroundThread =
          new Thread(new ThreadStart(p.PrintNumbers));
        backgroundThread.Name = "Secondary";
        backgroundThread.Start();
      break;
      case "1":
        p.PrintNumbers();
      break;
      default:
        Console.WriteLine("I don't know what you want...you get 1 thread.");
        goto case "1";
    }

    // 做其他一些工作
    MessageBox.Show("I'm busy!", "Work on main thread...");
    Console.ReadLine();
  }
```

如果以单线程运行这个程序，将发现直到全部的数字输出到了控制台上后，消息（对话框）才被显示出来。由于在每个数字输出来后都需要暂停大概两秒钟，所以这是一种非常糟糕的用户体验。然而，如果选择两个线程，消息对话框就会立刻显示出来，因为单独的Thread对象负责将数字输出到控制台上（如图19-2所示）。

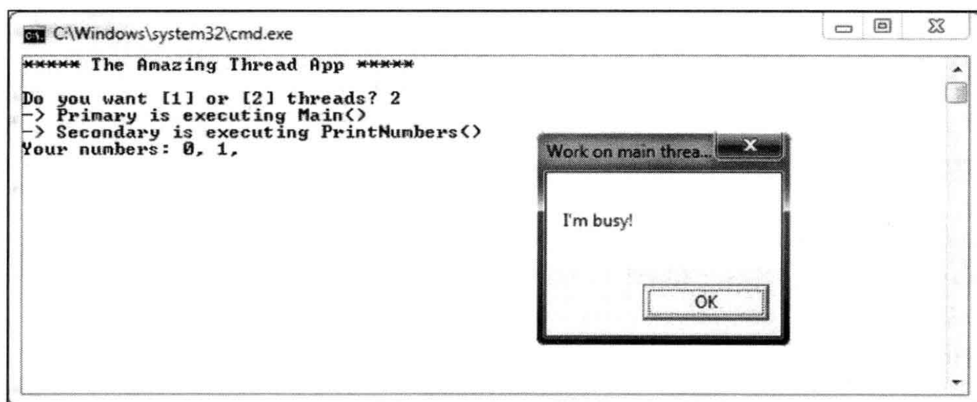


图19-2　多线程应用程序提供更多客户响应

源代码　SimpleMultiThreadApp项目的源代码位于Chapter 19子目录下。

19.7.2　使用 ParameterizedThreadStart 委托

回想一下，ThreadStart委托仅仅指向无返回值、无参数的方法。虽然这能满足大多数情况下的要求，但是，如果想把数据传递给在次线程上执行的方法，则需要使用ParameterizedThreadStart委托类型。在下面的例子中，本章前面已经建立了AsyncCallbackDelegate项目，我们将在此之上重新建立逻辑，这一次使用ParameterizedThreadStart委托类型。

首先，创建一个名为AddWithThreads的控制台程序，并引入System.Threading命名空间。由于ParameterizedThreadStart能够指向任意带一个System.Object参数的方法，所以要创建一个自定义类型，这个类型包含要增加的数字，如下所示：

```
class AddParams
{
  public int a, b;

  public AddParams(int numb1, int numb2)
  {
    a = numb1;
    b = numb2;
  }
}
```

接下来，在Program类中创建一个静态方法，这个方法使用AddParams类型作为参数，用于输出每次相加的结果，如下所示：

```
static void Add(object data)
{
  if (data is AddParams)
  {
    Console.WriteLine("ID of thread in Add(): {0}",
      Thread.CurrentThread.ManagedThreadId);

    AddParams ap = (AddParams)data;
    Console.WriteLine("{0} + {1} is {2}",
      ap.a, ap.b, ap.a + ap.b);
  }
}
```

Main()中的代码直截了当，使用ParameterizedThreadStart比使用ThreadStart更加简单，如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Adding with Thread objects *****");
  Console.WriteLine("ID of thread in Main(): {0}",
    Thread.CurrentThread.ManagedThreadId);

  // 建立AddParams对象，将其传给次线程
  AddParams ap = new AddParams(10, 10);
  Thread t = new Thread(new ParameterizedThreadStart(Add));
  t.Start(ap);

  // 强制等待以让其他线程结束
  Thread.Sleep(5);

  Console.ReadLine();
}
```

19.7.3 AutoResetEvent类

在之前的示例中，我们使用了一种简陋的方式来通知主线程等待，直到次线程结束。在学习异步委托时，我们使用了一个简单的bool变量作为开关，但这并不是推荐的解决方案，因为两个线程都能访问相同的数据点，并且这将导致数据损坏（data corruption）。一个较安全但仍然不可取的方法是调用Thread.Sleep()，等待一段固定的时间。但问题是你并不想等待过长的时间。

一个简单、线程安全的方法是使用AutoResetEvent类，强制线程等待，直到其他线程结束。在需要等待的线程中（如Main()方法），创建该类的实例，向构造函数传入false，表示尚未收到通知。然后在需要等待的地方调用WaitOne()方法。以下是修改后的Program类，它使用静态的AutoResetEvent成员变量进行这些处理：

```
class Program
{
  private static AutoResetEvent waitHandle = new AutoResetEvent(false);

  static void Main(string[] args)
  {
    Console.WriteLine("***** Adding with Thread objects *****");
    Console.WriteLine("ID of thread in Main(): {0}",
      Thread.CurrentThread.ManagedThreadId);
    AddParams ap = new AddParams(10, 10);
    Thread t = new Thread(new ParameterizedThreadStart(Add));
    t.Start(ap);

    // 等待，直到收到通知
    waitHandle.WaitOne();
    Console.WriteLine("Other thread is done!");

    Console.ReadLine();
  }
...
}
```

当其他线程完成任务时，将调用同一个AutoResetEvent类型实例的Set()方法：

```
static void Add(object data)
{
  if (data is AddParams)
  {
    Console.WriteLine("ID of thread in Add(): {0}",
      Thread.CurrentThread.ManagedThreadId);

    AddParams ap = (AddParams)data;
    Console.WriteLine("{0} + {1} is {2}",
      ap.a, ap.b, ap.a + ap.b);

    // 通知其他线程，该线程已结束
    waitHandle.Set();
  }
}
```

源代码 AddWithThreads项目的源代码位于Chapter 19子目录下。

19.7.4 前台线程和后台线程

了解如何使用System.Threading命名空间创建新的线程之后，下面来正式看一看前台线程和后台线程的区别。

- **前台线程**能阻止应用程序的终结。一直到所有的前台线程终止后，CLR才能关闭应用程序（即卸载承载的应用程序域）。

- ❑ **后台线程**（有时也叫守护线程，daemon thread）被CLR认为是程序执行中可做出牺牲的途径，即在任何时候（即使这个线程此时正在执行某项工作）都可能被忽略。因此，如果所有的前台线程终止，当应用程序域卸载时，所有的后台线程也会被自动终止。

值得重点注意的是，前台线程和后台线程并不等同于主线程和工作者线程。默认情况下，所有通过Thread.Start()方法创建的线程都自动成为前台线程。这意味着，直到所有的线程本身单元的工作都执行完成了，应用程序域才会卸载。大多情况下，这非常有必要。

但是，反过来设想一下，假定在一个后台线程上调用Printer.PrintNumbers()，这意味着，该方法通过Thread类型（ThreadStart或ParameterizedThreadStart委托）可以被安全地结束，而此时所有的前台线程可以做它们自己的工作。只要把IsBackground属性设为true就可以将线程配置为后台线程：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Background Threads *****\n");

  Printer p = new Printer();
  Thread bgroundThread =
    new Thread(new ThreadStart(p.PrintNumbers));
  // 这是后台线程
  bgroundThread.IsBackground = true;
  bgroundThread.Start();
}
```

请注意，Main()方法中没有Console.ReadLine()这条语句（该条语句能够保证在按Enter键之前，控制台窗口始终可见）。因此，由于Thread对象被定义成后台线程，当程序一运行就立即结束了。由于Main()方法触发建立主前台线程，所以一旦Main()执行完毕，应用程序域将会在次线程结束自身工作之前卸载。

但是，如果注释掉设置IsBackground属性的那行语句，所有数字都会被输出来，这是因为，在应用程序域从所承载的进程中卸载之前，所有的前台线程必须将它们的任务执行完成。

多数情况下，当程序的主任务完成，而工作者线程正在执行无关紧要的任务时，把工作线程配置成后台类型是很有用的。例如，构建一个每隔几分钟就ping一次邮件服务器看有没有新邮件的应用程序，或更新当前天气条件等其他无关紧要的任务。

19.8 并发问题

在构建多线程应用程序时，需要确保任何共享数据都处于被保护状态，以防止多个线程修改它的值。由于一个应用程序域中的所有线程都能够并发访问共享数据，所以，想象一下当它们正在访问其中的某个数据项时，会发生什么。由于线程调度器会随机挂起线程，所以如果线程A在完成之前被挂起了，线程B读到的就是一个不稳定的数据。

为举例说明并发的问题，构建一个名为MultiThreadedPrinting的C#控制台程序，这个程序使用了前面创建的Printer类，这一次PrintNumbers()方法将暂停每个线程，时间由随机数决定：

```
public class Printer
{
  public void PrintNumbers()
  {
...
```

```
      for (int i = 0; i < 10; i++)
      {
        // 使线程休眠数秒
        Random r = new Random();
        Thread.Sleep(1000 * r.Next(5));
        Console.Write("{0}, ", i);
      }
      Console.WriteLine();
    }
  }
```

Main()方法负责创建一个拥有10个（不同名称的）Thread对象的数组，并且每一个对象都调用Printer对象的同一个实例，如下所示：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("*****Synchronizing Threads *****\n");

    Printer p = new Printer();

    // 使10个线程全部指向同一对象的同一方法
    Thread[] threads = new Thread[10];
    for (int i = 0; i < 10; i++)
    {
      threads[i] =
        new Thread(new ThreadStart(p.PrintNumbers));
      threads[i].Name = string.Format("Worker thread #{0}", i);
    }

    // 现在开始每一个线程
    foreach (Thread t in threads)
      t.Start();
    Console.ReadLine();
  }
}
```

在看运行结果之前，我们来归纳一下问题：在应用程序域中的主线程产生了10个工作者线程，每一个工作者线程执行同一个Printer实例的PrintNumbers()方法。由于没有预防锁定共享资源（控制台），故在PrintNumbers()输出到控制台之前，调用PrintNumbers()方法的线程很有可能会被挂起。因为不知道挂起什么时候（或者是否有）可能发生，所以我们得到的是不可预测的结果。比如，可能得到下面的输出结果：

```
*****Synchronizing Threads *****

-> Worker thread #1 is executing PrintNumbers()
Your numbers: -> Worker thread #0 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
Your numbers: -> Worker thread #3 is executing PrintNumbers()
Your numbers: -> Worker thread #4 is executing PrintNumbers()
Your numbers: -> Worker thread #6 is executing PrintNumbers()
Your numbers: -> Worker thread #7 is executing PrintNumbers()
Your numbers: -> Worker thread #8 is executing PrintNumbers()
Your numbers: -> Worker thread #9 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #5 is executing PrintNumbers()
```

```
Your numbers: 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 2, 1, 0, 0, 4, 3,
4, 1, 2, 4, 5, 5, 5, 6, 6, 6, 2, 7, 7, 7, 3, 4, 0, 8, 4, 5, 1, 5, 8, 8, 9,
2, 6, 1, 0, 9, 1,
6, 2, 7, 9,
2, 1, 7, 8, 3, 2, 3, 3, 9,
8, 4, 4, 5, 9,
4, 3, 5, 5, 6, 3, 6, 7, 4, 7, 6, 8, 7, 4, 8, 5, 5, 6, 6, 8, 7, 7, 9,
8, 9,
8, 9,
9,
9,
```

现在把程序运行几遍。下面显示了一种可能的结果（读者的运行结果可能明显不同）。

```
*****Synchronizing Threads *****

-> Worker thread #0 is executing PrintNumbers()
-> Worker thread #1 is executing PrintNumbers()
-> Worker thread #2 is executing PrintNumbers()
Your numbers: -> Worker thread #4 is executing PrintNumbers()
Your numbers: -> Worker thread #5 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #6 is executing PrintNumbers()
Your numbers: -> Worker thread #7 is executing PrintNumbers()
Your numbers: Your numbers: -> Worker thread #8 is executing PrintNumbers()
Your numbers: -> Worker thread #9 is executing PrintNumbers()
Your numbers: -> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7
, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9,
9,
9,
9,
9,
9,
9,
9,
9,
9,
```

说明 如果（上面例子中）无法得到随机的输出，请把线程的个数从10变到100，或调用Thread.Sleep()方法。最终会遇到并发的问题。

显然这里有问题。当每个线程都调用Printer来输出数字的时候，线程调度器可能正在切换线程。这导致了不同的输出结果。我们需要找到一种方式来通过编程控制对共享资源的同步访问。同读者猜想的一样，System.Threading命名空间提供了一些以同步为中心的类型。C#编程语言也提供了一个特别的关键字，它能在多线程程序中同步共享数据。

19.8.1 使用C#的lock关键字进行同步

同步访问共享资源的首选技术是C#的lock关键字。这个关键字允许定义一段线程同步的代码语句。采用这项技术，后进入的线程不会中断当前线程，而是停止自身下一步执行。lock关键字需要定义一个标记（即一个对象引用），线程在进入锁定范围的时候必须获得这个标记。当试图锁定的是一个实例级对象的私有方法时，使用方法本身所在对象的引用就可以了，如下所示：

```
private void SomePrivateMethod()
{
  // 使用当前对象作为线程标记
  lock(this)
  {
    // 所有在这个范围内的代码是线程安全的
  }
}
```

然而，如果需要锁定公共成员中的一段代码，比较安全（也比较推荐）的方式是声明私有的object成员来作为锁标识，如下所示：

```
public class Printer
{
  // 锁标识
  private object threadLock = new object();

  public void PrintNumbers()
  {
    // 使用锁标识
    lock (threadLock)
    {
      ...
    }
  }
}
```

如果分析PrintNumbers()方法，可以看到线程强占的共享资源是控制台窗口，因此，所有和Console类型交互的代码都必须在锁定范围中。

```
public void PrintNumbers()
{
  // 使用私有对象锁定标记
  lock (threadLock)
  {
    // 显示线程信息
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);

    // 输出数字
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
    Console.WriteLine();
  }
}
```

现在已经有效设计了一个保证当前线程完成任务的方法。一旦一个线程进入锁定范围，在它退出锁定范围且释放锁定之前，其他线程将无法访问锁定标记（本例中是当前对象的引用）。因此，如果线程A获得锁定标记，直到它放弃这个锁定标记，其他线程才能够进入锁定范围。

说明 如果试图锁定静态方法中的代码，只需要声明一个私有静态对象成员变量作为锁定标记就可以了。

如果运行这个程序，我们将看到每个线程都能够完成它们的相关逻辑，而不被中断：

```
*****Synchronizing Threads *****

-> Worker thread #0 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #1 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #3 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #2 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #4 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #5 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #7 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #6 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #8 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
-> Worker thread #9 is executing PrintNumbers()
Your numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

源代码 MultiThreadedPrinting项目的源代码位于Chapter 19子目录下。

19.8.2 使用System.Threading.Monitor类型进行同步

C# lock声明实际上是和System.Threading.Monitor类一同使用时的速记符号。经过编译器的处理，锁定区域实际上被转化成了如下的内容（可以使用ildasm.exe进行验证）：

```
public void PrintNumbers()
{
  Monitor.Enter(threadLock);
  try
  {
    // 显示线程信息
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);
```

```
      // 输出数字
      Console.Write("Your numbers: ");
      for (int i = 0; i < 10; i++)
      {
        Random r = new Random();
        Thread.Sleep(1000 * r.Next(5));
        Console.Write("{0}, ", i);
      }
      Console.WriteLine();
    }
    finally
    {
      Monitor.Exit(threadLock);
    }
  }
```

首先，请注意Monitor.Enter()方法是线程标记的最终容器，而该线程标记作为参数由用户指定给lock关键字。接下来，所有在锁定范围中的代码被try块包含。而对应的finally子句保证了无论出现什么运行错误，线程标记都能被释放（通过Monitor.Exit()方法）。如果修改MultiThreadPrinting程序以便直接使用Monitor类型（像上面显示的那样），看到的结果是一样的。

既然使用lock关键字比使用System.Threading.Monitor类型的代码更少，那么直接使用Monitor类型的好处何在？一句话——更好的控制能力。使用Monitor类型，可以（使用Monitor.Wait()方法）指示活动的线程等待一段时间，在当前线程完成操作时，（使用Monitor.Pulse()或Monitor.PulseAll()）通知等待中的线程。

同读者预测的一样，在大多数情况下，C# lock关键字就够用了。如果对Monitor的其他成员感兴趣的话，请参考.NET Framework 4.5 SDK文档。

19.8.3 使用System.Threading.Interlocked类型进行同步

虽然这很难相信，但如果看了底层的CIL代码，将发现赋值和简单的数值运算都不是原子型操作。由此，System.Threading命名空间提供了一个类型允许我们来原子型操作单个数据，使用它比使用Monitor类型更加简单。Interlocked类定义如表19-4所示的静态成员。

表19-4 System.Threading.Interlocked类型的部分静态成员

成 员	作 用
CompareExchange()	安全地比较两个值是否相等。如果相等，将第3个值与其中1个值交换
Decrement()	安全递减1
Exchange()	安全地交换数据
Increment()	安全递加1

虽然不太起眼，但是原子型地修改单个值在多线程环境下非常普遍。假设有个方法名为AddOne()，它用来给名为intVal的整型变量加1，写出如下的同步代码：

```
public void AddOne()
{
  lock(myLockToken)
  {
```

```
      intVal++;
    }
  }
```

可以通过静态的Interlocked.Increment()方法简化代码；以引用方式传入要递增的变量。注意，Increment()方法不但可以修改传入的参数值，还会返回递增后的新值：

```
public void AddOne()
{
  int newVal = Interlocked.Increment(ref intVal);
}
```

除了Increment()和Decrement()，使用Interlocked类型还可以原子型地赋值给数字或对象。例如，如果想把83赋给一个成员变量，无须明确使用lock声明（或Monitor逻辑），使用Interlocked.Exchange()方法就可以了，如下所示：

```
public void SafeAssignment()
{
  Interlocked.Exchange(ref myInt, 83);
}
```

最后，如果想通过在线程安全的情况下测试两个值是否相等来改变比较后的指向，可以像下面这样调用Interlocked.CompareExchange()方法：

```
public void CompareAndExchange()
{
  // 如果i等于83，把99赋给i
  Interlocked.CompareExchange(ref i, 99, 83);
}
```

19.8.4 使用[Synchronization]特性进行同步

最后一个同步化原语是[Synchronization]特性。它位于System.Runtime.Remoting.Contexts命名空间下。这个类级别的特性有效地使对象的所有实例的成员都保持线程安全。当CLR分配带[Synchronization]的对象时，它会把这个对象放在同步上下文中。回想一下在第17章里，要想对象不被在上下文边界中移动，就必须让它继承ContextBoundObject类。因此，如果想要使Printer类类型线程安全（在类成员中不使用明确的线程安全代码），可以做如下修改：

```
using System.Runtime.Remoting.Contexts;
...

// Printer的全部方法都是线程安全的
 [Synchronization]
public class Printer : ContextBoundObject
{
  public void PrintNumbers()
  {
    ...
  }
}
```

在有些方面，像这样写线程安全的代码是一种“偷懒的”方式，因为它不需要我们实际深入线程控制敏感数据的细节。这种方式的主要问题是：即使一个方法没有使用线程敏感的数据，CLR仍然会锁定对该方法的调用。很明显，这会全面降低性能，所以要小心使用这种方式。

19.9 使用 TimerCallback 编程

许多程序需要定期调用具体的方法。比如，可能有一个应用程序需要在状态栏上通过一个辅助函数显示当前的时间，或者，可能希望应用程序调用一个辅助函数，让它经常执行非紧迫的后台任务，比如检查是否收到新邮件。像这些情况，就可以使用System.Threading.Timer类型和与其相关的TimerCallback委托。

为举例说明，设想有一个控制台应用程序（TimerApp），这个程序每隔1秒把当前时间输出来，直到用户按键中断这个程序。很明显，第一步我们要写这个输出的方法，这个方法将会被Timer类型调用(确保在代码文件中导入了System.Threading)：

```
class Program
{
  static void PrintTime(object state)
  {
    Console.WriteLine("Time is: {0}",
      DateTime.Now.ToLongTimeString());
  }

  static void Main(string[] args)
  {
  }
}
```

注意PrintTime()方法拥有一个System.Object类型的参数并且返回void。这里没有其他的选择，因为TimerCallback委托仅仅调用符合这样的签名的方法。传入TimerCallback委托的参数可以是任何对象类型（在邮件示例程序中，参数可能表示在这个过程期间发生交互的Microsoft Exchange服务器的名字）。还需要注意的是，由于这个参数是System.Object类型，所以可以使用System.Array或者自定义类/结构传入多个值。

下一步，定义一个TimerCallback委托实例，并把它传入Timer对象中。除了定义TimerCallback委托，Timer的构造函数还允许定义别的信息传送到委托指向的方法中：一个可选参数（定义为System.Object类型）、访问时间间隔，以及在第一次调用之前等待多长时间（以毫秒为单位）。举例如下：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Working with Timer type *****\n");

  // 为Timer类型创建委托
  TimerCallback timeCB = new TimerCallback(PrintTime);

  // 设置Timer类
  Timer t = new Timer(
    timeCB,     // TimerCallback委托对象
    null,       // 想传入的参数 (null表示没有参数)
    0,          // 在开始之前，等待多长时间 (以毫秒为单位)
    1000);      // 每次调用的间隔时间 (以毫秒为单位)

  Console.WriteLine("Hit key to terminate...");
  Console.ReadLine();
}
```

本例中，PrintTime()方法大概每隔一秒钟调用一次，并且没有什么信息需要传入。下面是输出结果：

```
***** Working with Timer type *****

Hit key to terminate...
Time is: 6:51:48 PM
Time is: 6:51:49 PM
Time is: 6:51:50 PM
Time is: 6:51:51 PM
Time is: 6:51:52 PM
Press any key to continue . . .
```

如果希望传递一些信息给委托指向的方法，把Timer构造函数的第二个参数用指定值取代空值就可以了，如下所示：

```
// 设置timer类
Timer t = new Timer(timeCB, "Hello From Main", 0, 1000);
```

可以这样来获得传入的数据：

```
static void PrintTime(object state)
{
  Console.WriteLine("Time is: {0}, Param is: {1}",
    DateTime.Now.ToLongTimeString(), state.ToString());
}
```

源代码 TimerApp项目的源代码位于Chapter 19子目录下。

19.10 CLR 线程池

最后关于线程的核心主题是CLR线程池。当使用委托类型（通过BeginInvoke()方法）进行异步方法调用的时候，CLR并不会创建新的线程。为了取得更高的效率，委托的BeginInvoke()方法创建了由运行时维护的工作者线程池。为了更好地和这些线程进行交互，System.Threading命名空间提供了ThreadPool类类型。

如果想使用池中的工作者线程排队执行一个方法，可以使用ThreadPool.QueueUserWorkItem()方法。这个方法进行了重载，除了可以传递一个WaitCallback委托之外还可以指定一个可选的表示自定义状态数据的System.Object：

```
public static class ThreadPool
{
  ...
  public static bool QueueUserWorkItem(WaitCallback callBack);
  public static bool QueueUserWorkItem(WaitCallback callBack,
    object state);
}
```

WaitCallback委托指向有单个System.Object类型的参数（代表了可选的状态数据）且无返回值的

方法。注意，如果在调用QueueUserWorkItem()时不提供这个参数，CLR会自动传送null值。为举例说明CLR线程池中使用的队列方法，仔细思考下面的程序。它再次使用了前面的Printer类型，但是，在这个例子中，不需要手工创建Thread对象数组，只要把线程池的成员指向PrintNumbers()方法就可以了。

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with the CLR Thread Pool *****\n");

    Console.WriteLine("Main thread started. ThreadID = {0}",
      Thread.CurrentThread.ManagedThreadId);

    Printer p = new Printer();

    WaitCallback workItem = new WaitCallback(PrintTheNumbers);

    // 调用这个方法10次
    for (int i = 0; i < 10; i++)
    {
      ThreadPool.QueueUserWorkItem(workItem, p);
    }
    Console.WriteLine("All tasks queued");
    Console.ReadLine();
  }

  static void PrintTheNumbers(object state)
  {
    Printer task = (Printer)state;
    task.PrintNumbers();
  }
}
```

现在，读者可能想知道比起显式创建Thread对象，使用这个被CLR所维护的线程池的好处何在。我认为使用线程池的主要好处是：

- 线程池减少了线程创建、开始和停止的次数，而这提高了效率；
- 使用线程池，能够使我们将注意力放到业务逻辑上而不是多线程架构上。

然而，在某些情况下应优先使用手工线程管理，具体如下。

- 如果需要前台线程或设置优先级别。线程池中的线程总是后台线程，且它的优先级是默认的（ThreadPriority.Normal）。
- 如果需要有一个带有固定标识的线程便于退出、挂起或通过名字发现它。

源代码　ThreadPoolApp项目的源代码位于Chapter 19子目录下。

System.Threading命名空间的学习至此告一段落。在创建多线程应用程序时，理解本章目前所介绍的这些话题（特别是并行问题）绝对是非常有价值的。有了这些基础，下面我们将注意力转到一些新的与线程相关的话题，它们只能在.NET 4.0或更高版本下使用。首先介绍的是另一种线程模型——TPL。

19.11　使用任务并行库进行并行编程

截至目前，本章介绍了两种构建多线程软件的编程技术（使用异步委托或通过System.Threading的成员）。记住，这两种方法可以在任何版本的.NET平台下工作。

从.NET 4.0开始，微软引入了一种全新的多线程应用程序开发方法，即使用TPL并行编程库。使用System.Threading.Tasks中的类型，可以构建细粒度的、可扩展的并行代码，而不必直接与线程和线程池打交道。

但这并不是说再使用TPL的时候不会用到System.Threading。实际上，这两种线程工具可以非常自然地一起工作。特别是因为System.Threading命名空间还提供了我们之前介绍过的大量同步基础组件（Monitor、Interlocked等）。也就是说，你很可能会发现你更喜欢使用TPL，而不是原始的System.Threading命名空间，因为使用TPL，同样的任务可以以更加直接的方式执行。

说明　注意，新的.NET 4.5 C# async和await关键字使用了大量System.Threading.Tasks命名空间中的成员。

19.11.1　任务并行库API

总体而言，System.Threading.Tasks中的类型被称为任务并行库（Task Parallel Library，TPL）。TPL使用CLR线程池自动将应用程序的工作动态分配到可用的CPU中。TPL还处理工作分区、线程调度、状态管理和其他低级别的细节操作。最终结果是，你可以最大限度地提升.NET应用程序的性能，并且避免直接操作线程所带来的复杂性（如图19-3所示）。

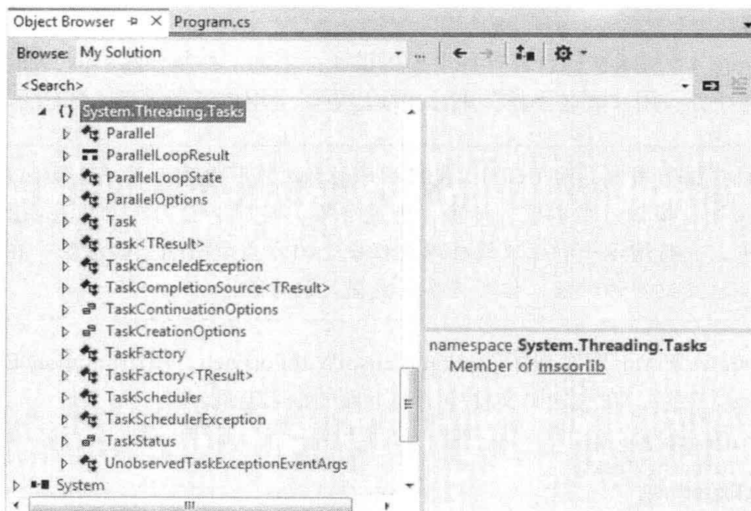


图19-3　System.Threading.Tasks命名空间的成员

19.11.2　Parallel类的作用

TPL中一个十分重要的类是System.Threading.Tasks.Parallel，它提供了大量方法，能够以并行的方式迭代数据集合（实现了IEnumerable<T>的对象）。在.NET Framework 4.5 SDK文档中查看Parallel类，你会发现该类支持两个主要的静态方法——Parallel.For()和Parallel.ForEach()，每个方法都有很多重载版本。

这些方法可用于编写并行执行的代码体。从概念上说，这些语句的逻辑与普通循环（使用for或foreach关键字）中的逻辑完全相同。不过好处是，Parallel类将从线程池中为我们提取线程（和管理并发）。

这两个方法都要求指定一个与IEnumerable或IEnumerable<T>兼容的容器，来保存需要并行处理的数据。容器可以是简单的数组、非泛型集合（如ArrayList）、泛型集合（如List<T>）或LINQ查询的结果。

此外，你还需要使用System.Func<T>和System.Action<T>委托来指定要调用的处理数据的方法。我们在第12章中介绍LINQ to Object时已经学习了Func<T>委托。它表示一个拥有给定返回值和不同数量参数的方法。Action<T>委托与Func<T>类似，指向有几个参数的方法。但是Action<T>只能返回void。

尽管在调用Parallel.For()和Parallel.ForEach()方法时可以传递强类型的Func<T>或Action<T>委托对象，你也可以使用恰当的C#匿名方法或Lambda表达式来简化编程。

19.11.3　使用Parallel类的数据并行

使用TPL的第一种方式是执行数据并行（data parallelism）。简单地说，该术语是指使用Parallel.For()或Parallel.ForEach()方法以并行方式对数组或集合中的数据进行迭代。假设需要执行一些大工作量的文件IO操作。如需要向内存中加载大量*.jpg文件并进行翻转，然后将修改后的图像数据保存到新的位置。

.NET Framework 4.5 SDK文档针对这种情形提供了一个基于控制台的示例，而我们要用图形用户界面来执行相同的任务，用“匿名委托”来让次线程更新主用户界面线程（即UI线程）。

说明　在构建多线程图形用户界面（GUI）应用程序时，次线程永远无法直接访问用户界面控件。原因是这些控件（按钮、文本框、标签、进度条等）与创建它们的线程存在线程相关性。在下面的示例中，我将演示一种让次线程以线程安全的方式访问UI项的途径。在介绍.NET 4.5 C# async和await关键字的时候，你还将看到更简单的方法。

创建一个Windows Forms应用程序DataParallelismWithForEach，使用Solution Explorer将Form1.cs改名为MainForm.cs。然后，在主代码文件中引入下面的命名空间：

```
// 确保引入下面这些命名空间
using System.Threading.Tasks;
using System.Threading;
using System.IO;
```

该应用程序的GUI包含一个多行的TextBox和一个Button（名为btnProcessImages）。当任务在后台

执行时，你可以向文本区域输入数据，以此来演示并行任务的无阻塞性。该Button的Click事件最终将使用TPL，但现在先使用下面的阻塞代码。

说明 你应该修改传入给Directory.GetFiles()方法调用的字符串，让它指向电脑中包含图片文件的路径（如存放家庭照片的个人文件夹）。这里我只指向了存放在C:\Users\Public\Pictures\Sample Pictures下的一些图片。

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }

  private void btnProcessImages_Click(object sender, EventArgs e)
  {
    ProcessFiles();
  }

  private void ProcessFiles()
  {
    // 加载所有*.jpg文件，并为修改后的数据新建一个文件夹
    string[] files = Directory.GetFiles
      (@"C:\Users\Public\Pictures\Sample Pictures", "*.jpg",
      SearchOption.AllDirectories);
    string newDir = @"C:\ModifiedPictures";
    Directory.CreateDirectory(newDir);

    // 以阻塞方式处理图像数据
    foreach (string currentFile in files)
    {
      string filename = Path.GetFileName(currentFile);

      using (Bitmap bitmap = new Bitmap(currentFile))
      {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(newDir, filename));

        // 打印处理当前图像的线程ID
        this.Text = string.Format("Processing {0} on thread {1}", filename,
          Thread.CurrentThread.ManagedThreadId);
      }
    }
  }
}
```

注意，ProcessFiles()方法将翻转指定目录下的所有37个*.jpg文件（再次强调，必要时可以更改Directory.GetFiles()方法的路径）。现在，所有的工作都发生在程序的主线程上。因此单击按钮将挂起程序。此外，窗口标题将显示正在处理文件的为相同的主线程，因为我们只有一个执行线程。

为了让文件在更多的CPU中处理，可以使用Parallel.ForEach()来代替当前的foreach循环。该方法有多个重载版本，不过最简单的版本只要求指定一个与IEnumerable<T>兼容的包含待处理数据的对象（文件字符串数组）和一个用于指向执行工作的方法的Action<T>委托。

以下是相关的修改，其中使用了C#的Lambda操作符来代替字面的Action<T>委托对象。注意，我们现在注释掉了显示执行当前图像文件的线程ID。原因到下一节再揭晓。

```
// 以并行方式处理图像数据
Parallel.ForEach(files, currentFile =>
  {
    string filename = Path.GetFileName(currentFile);

    using (Bitmap bitmap = new Bitmap(currentFile))
    {
      bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
      bitmap.Save(Path.Combine(newDir, filename));

      // 下面的语句现在有问题！参见下一节
      // this.Text = string.Format("Processing {0} on thread {1}", filename,
      // Thread.CurrentThread.ManagedThreadId);
    }
  }
);
```

19.11.4　在次线程中访问UI元素

你会发现我注释了上面用当前执行线程的ID更新主窗体标题的代码行。如上所述，GUI控件与创建它的线程具有“线程相关性”。如果次线程试图访问不是由它直接创建的控件，在调试软件时一定会遇到运行时错误。另一方面，如果运行程序（通过Ctrl+F5），原始代码可能永远不会产生任何错误。

说明　重申一下：在调试（F5）多线程应用程序时，当次线程“触摸”主线程创建的控件时，Visual Studio通常会捕获到因此引发的错误。但是，在运行（Ctrl+F5）应用程序时，往往会运行正确（当然也可能会出错）。如果不采取防范措施（下面会介绍），你的应用程序在这种条件下会存在引发运行时错误的可能。

我们可以使用一种方法来让次线程以线程安全的方式访问这些控件，即用另一种与委托相关的技术——匿名委托。Windows Forms API中的Control父类定义了一个Invoke()方法，接受一个System.Delegate作为参数。你可以在次线程的上下文中调用该方法，可以线程安全地更新给定控件的UI。如今，尽管我们可以直接编写所有需要的委托代码，但大多数开发者还是选择简单地使用匿名委托。下面的代码相应地修改了上面注释的代码：

```
using (Bitmap bitmap = new Bitmap(currentFile))
{
  bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
  bitmap.Save(Path.Combine(newDir, filename));

  //呃！不能再用了！
  //this.Text = string.Format("Processing {0} on thread {1}", filename,
  // Thread.CurrentThread.ManagedThreadId);

  //在Form对象上调用，允许次线程以线程安全的方式访问控件
  this.Invoke((Action)delegate
    {
```

```
            this.Text = string.Format("Processing {0} on thread {1}", filename,
                        Thread.CurrentThread.ManagedThreadId);
        }
      );
    }
```


说明 this.Invoke()方法只在Windows Form API中存在。在构建WPF应用程序时，可以使用this.Diepatcher.Invoke()来达到同样的目的。

现在运行程序，TPL会将工作分配给线程池中的多个线程，使用尽可能多的CPU。但是，在所有图像处理完毕之前，你无法看到窗口标题显示每个线程的名称，也无法看到在文本框中输入的内容。这是因为主UI线程仍然被阻塞，等待所有线程完成它们的工作。

19.11.5 Task类

Task类可以轻松地在次线程中调用方法，可以作为异步委托的简单替代品。更新Button控件的Click处理程序：

```
private void btnProcessImages_Click(object sender, EventArgs e)
{
  // 启动一个新的“任务”来处理文件
  Task.Factory.StartNew(() =>
  {
     ProcessFiles();
  });
}
```

Task的Factory属性返回一个TaskFactory对象。在调用StartNew()方法时，传入一个Action<T>委托（这里为一个Lambda表达式），指向以异步方式进行调用的方法。经过这个小的修改，你会发现窗口的标题可以显示线程池中正在处理文件的线程名称，更棒的是，文本区域也可以接收输入了，因为UI线程已经不再是阻塞的了。

19.11.6 处理取消请求

可以对当前示例作一个改进，让用户可以通过一个Cancel按钮停止处理图像数据。幸好Parallel.For()和Parallel.ForEach()方法都支持取消标记。调用Parallel的方法时，可以传入一个ParallelOptions对象，它包含一个CancellationTokenSource对象。

首先，在Form派生类中定义下面的CancellationTokenSource类型的私有成员变量cancelToken：

```
public partial class MainForm : Form
{
  // 新的Form级别的变量
  private CancellationTokenSource cancelToken =
    new CancellationTokenSource();
...
}
```

现在，假定已经在设计器上添加了一个新的Button（btnCancel），处理其Click事件，并实现处理程序：

```
private void btnCancel_Click(object sender, EventArgs e)
```

```
{
  // 停止所有工作者线程
  cancelToken.Cancel();
}
```

真正的修改发生在ProcessFiles()方法中。考虑下面的最终实现：

```
private void ProcessFiles()
{
  // 使用ParallelOptions实例保存CancellationToken
  ParallelOptions parOpts = new ParallelOptions();
  parOpts.CancellationToken = cancelToken.Token;
  parOpts.MaxDegreeOfParallelism = System.Environment.ProcessorCount;

  // 加载所有*.jpg文件，并为修改后的数据新建一个文件夹
  string[] files = Directory.GetFiles
    (@"C:\Users\Public\Pictures\Sample Pictures", "*.jpg",
        SearchOption.AllDirectories);
  string newDir = @"C:\ModifiedPictures";
  Directory.CreateDirectory(newDir);

  try
  {
    // 以并行方式处理图像数据
    Parallel.ForEach(files, parOpts, currentFile =>
    {
      parOpts.CancellationToken.ThrowIfCancellationRequested();

      string filename = Path.GetFileName(currentFile);
      using (Bitmap bitmap = new Bitmap(currentFile))
      {
        bitmap.RotateFlip(RotateFlipType.Rotate180FlipNone);
        bitmap.Save(Path.Combine(newDir, filename));
        this.Invoke((Action)delegate
          {
            this.Text = string.Format("Processing {0} on thread {1}", filename,
            Thread.CurrentThread.ManagedThreadId);
          }
        );
      }
    }
    );
  }
  catch (OperationCanceledException ex)
  {
this.Invoke((Action)delegate
      {        this.Text = ex.Message;
      });
  }
}
```

注意，在方法的开始我们配置了ParallelOptions对象，使用CancellationTokenSource标记设置其CancellationToken属性。还要注意，在调用Parallel.ForEach()方法时，将ParallelOptions对象作为第二个参数。

在循环逻辑的作用域内，我们调用了标记的ThrowIfCancellationRequest()，它将确保在用户单击Cancel按钮时，所有的线程都会停止，并抛出一个运行时异常进行通知。捕获这个Operation CanceledException错误，将错误消息显示在主窗口的文本字段中。

源代码 DataParallelismWithForEach项目的源代码位于Chapter 19子目录下。

19.11.7 使用并行类的任务并行

除了数据并行，TPL还可以使用Parallel.Invoke()方法轻松地触发多个异步任务。这种方法比使用委托或System.Threading成员要更直接，但如果想要对任务的执行方式有更多的控制，可以放弃Parallel.Invoke()，而像前面的示例那样直接使用Task类。

为了演示任务并行，新建一个Windows Forms应用程序MyEBookReader，并确保引入System.Threading.Tasks和System.Net命名空间。这个示例改编自.NET Framework 4.5 SDK文档中的一个有用示例，它从Project Gutenberg（www.gutenberg.org）中获取公开可用的电子书，然后并行地执行一些复杂的任务。

该GUI包含一个多行的TextBox控件（txtBook）和两个Button控件（btnDownload和btnGetStats）。设计好UI之后，处理每个Button的Click事件，并在窗体的代码文件中声明一个类级别的string变量theEBook。btnDownload的Click处理程序的实现如下：

```
private void btnDownload_Click(object sender, EventArgs e)
{
  WebClient wc = new WebClient();
  wc.DownloadStringCompleted += (s, eArgs) =>
  {
    theEBook = eArgs.Result;
    txtBook.Text = theEBook;
  };

  // Project Gutenberg中的电子书，查尔斯·狄更斯所著的《双城记》
  wc.DownloadStringAsync(new Uri("http://www.gutenberg.org/files/98/98-8.txt"));
}
```

WebClient类是System.Net中的成员。该类提供了大量方法，可以向一个由URI标识的资源发送或接收数据。实际上，这些方法中的大多数都有异步版本，如DownloadStringAsync()。该方法将自动从线程池中获取一个新的线程。当WebClient获取完数据之后，将触发DownloadStringCompleted事件，我们已经在此使用C# Lambda表达式处理了该事件。如果调用该方法（DownloadString()）的异步版本，窗体将在相当长的一段时间内失去响应。

btnGetStats按钮控件的Click事件处理程序将提取theEBook变量中包含的单词，然后将字符串数组传递给辅助函数进行处理，如下所示：

```
private void btnGetStats_Click(object sender, EventArgs e)
{
  // 从电子书中获取单词
  string[] words = theEBook.Split(new char[]
    { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
    StringSplitOptions.RemoveEmptyEntries);

  // 找到最常用的10个单词
  string[] tenMostCommon = FindTenMostCommon(words);

  // 获取最长的单词
  string longestWord = FindLongestWord(words);
```

```
    // 完成所有任务之后，创建一个字符串以在消息框中显示所有的统计信息
    StringBuilder bookStats = new StringBuilder("Ten Most Common Words are:\n");
    foreach (string s in tenMostCommon)
    {
      bookStats.AppendLine(s);
    }
    bookStats.AppendFormat("Longest word is: {0}", longestWord);
    bookStats.AppendLine();
    MessageBox.Show(bookStats.ToString(), "Book info");
  }
```

FindTenMostCommon()方法使用LINQ查询获取一个列表，其中包含string数组中出现最频繁的string对象。FindLongestWord()获取最长的单词：

```
private string[] FindTenMostCommon(string[] words)
{
    var frequencyOrder = from word in words
                         where word.Length > 6
                         group word by word into g
                         orderby g.Count() descending
                         select g.Key;

    string[] commonWords = (frequencyOrder.Take(10)).ToArray();
    return commonWords;
}

private string FindLongestWord(string[] words)
{
    return (from w in words orderby w.Length descending select w).FirstOrDefault();
}
```

运行该项目，执行所有任务所消耗的时间与计算机中CPU的数量和处理器的整体速度有关。最终，将看到如图19-4所示的输出结果。

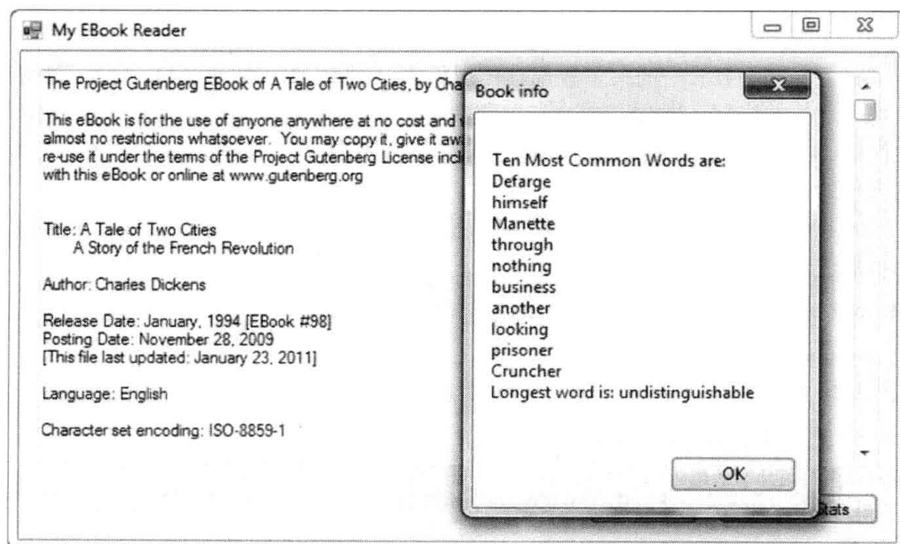


图19-4 所下载电子书的统计信息

你可以并行地调用FindTenMostCommon()和FindLongestWord()方法，让应用程序使用所有计算机中可用的CPU。修改btnGetStats_Click()方法，具体如下：

```
private void btnGetStats_Click(object sender, EventArgs e)
{
  // 从电子书中获取单词
  string[] words = theEBook.Split(
    new char[] { ' ', '\u000A', ',', '.', ';', ':', '-', '?', '/' },
    StringSplitOptions.RemoveEmptyEntries);
  string[] tenMostCommon = null;
  string longestWord = string.Empty;

  Parallel.Invoke(
    () =>
    {
      // 找到最常用的10个单词
      tenMostCommon = FindTenMostCommon(words);
    },
    () =>
    {
      // 获取最长的单词
      longestWord = FindLongestWord(words);
    });

  // 完成所有任务之后，创建一个字符串以在消息框中显示所有的统计信息
  ...
}
```

Parallel.Invoke()方法要求一个Action<>委托的参数数组，我们间接地使用Lambda表达式来提供。尽管输出结果相同，但TPL的好处是，所有计算机中可用的处理器都用来并行地调用各个方法。

源代码 MyEBookReader项目的源代码位于Chapter 19子目录下。

19.12 并行 LINQ 查询（PLINQ）

在学习TPL的最后，要注意还有一种在.NET应用程序里使用并行任务的方法。你可以使用一组扩展方法，构造LINQ查询来并行地执行工作。为并行运行而设计的LINQ查询称为PLINQ查询。

与使用Parallel类编写的并行代码类似，PLINQ也可以在需要时选择忽略并行处理集合的请求。PLINQ框架进行了很多优化，其中包括判断查询是否在同步方式下会运行得更快。

在运行时，PLINQ分析查询的整个结构，如果查询能从并行化中受益，则将同时运行。而如果并行执行查询会损害性能，PLINQ将按顺序运行查询。对于昂贵的并行算法和廉价的顺序算法，PLINQ默认会选择后者。

System.Linq命名空间的ParallelEnumerable类中包含了必要的扩展方法。表19-5列出了一些有用的PLINQ扩展。

要在实战中演示PLINQ，创建一个Windows Forms应用程序PLINQDataProcessingWithCancellation，引入System.Threading命名空间。这个简单的窗体仅需要两个Button控件——btnExecute和btnCancel。单击Execute按钮时，将启动一个Task，它对一个非常大的整型数组执行LINQ查询，查找x % 3 == 0为true的项。以下是该查询的非并行版本。

表19-5　ParallelEnumerable类中的部分成员

成　员	作　用
AsParallel()	指明如果可能的话，查询的其余部分应该并行化
WithCancellation()	指明PLINQ应该定期监视取消标记的状态，并在需要的时候取消执行
WithDegreeOfParallelism()	指明PLINQ进行并行查询时能使用的最大处理器数
ForAll()	使用foreach关键字枚举LINQ结果时进行并行处理，不必向消费者线程返回合并的结果

```
public partial class MainForm : Form
{
...
  private void btnExecute_Click(object sender, EventArgs e)
  {
    // 开始一个新的“任务”来处理整数
    Task.Factory.StartNew(() =>
    {
      ProcessIntData();
    });
  }

  private void ProcessIntData()
  {
    // 获得一个非常大的整数数组
    int[] source = Enumerable.Range(1, 10000000).ToArray();

    // 找到num % 3 == 0为true的数字，降序返回
    int[] modThreeIsZero = (from num in source where num % 3 == 0
          orderby num descending select num).ToArray();

    MessageBox.Show(string.Format("Found {0} numbers that match query!",
      modThreeIsZero.Count()));
  }
}
```

19.12.1　使用 PLINQ 查询

如果希望TPL并行地执行该查询（如果可以的话），可以像下面这样使用AsParallel()扩展方法：

```
int[] modThreeIsZero = (from num in source.AsParallel() where num % 3 == 0
                        orderby num descending select num).ToArray();
```

注意，LINQ查询的整个格式与前几章中看到的完全一样。但通过调用AsParallel()，TPL将尝试将工作传递给可用的CPU。

19.12.2　取消 PLINQ 查询

还可以使用CancellationTokenSource对象通知PLINQ查询，在条件满足时（通常是由于用户的干预）停止处理。声明一个窗体级别的CancellationTokenSource对象cancelToken，实现btnCancel的Click处理程序，调用该对象的Cancel()方法。以下是相关的代码：

```
public partial class MainForm : Form
{
  private CancellationTokenSource cancelToken = new CancellationTokenSource();
  private void btnCancel_Click(object sender, EventArgs e)
  {
    cancelToken.Cancel();
  }
...
}
```

现在，链式地调用WithCancellation()扩展方法并传入标记，就可以向PLINQ查询中引入取消请求。此外，还要将PLINQ查询包装在恰当的try/catch作用域内，并处理可能的异常。以下是ProcessIntData()方法的最终版本：

```
private void ProcessIntData()
{
  // 获得一个非常大的整数数组
  int[] source = Enumerable.Range(1, 10000000).ToArray();

  // 找到num % 3 == 0为true的数字，降序返回
  int[] modThreeIsZero = null;
  try
  {
    modThreeIsZero = (from num in
      source.AsParallel().WithCancellation(cancelToken.Token)
      where num % 3 == 0 orderby num descending
      select num).ToArray();
    MessageBox.Show(string.Format("Found {0} numbers that match query!",
      modThreeIsZero.Count()));
  }
  catch (OperationCanceledException ex)
  {
this.Invoke((Action)delegate
{
this.Text = ex.Message;
    });
  }
}
```

源代码 PLINQDataProcessingWithCancellation项目的源代码位于Chapter 19子目录下。

19.13 .NET 4.5 下的异步调用

我们在本章（相当长的一章）中已经介绍了很多简单的元素。毫无疑问，构建、调试和理解复杂的多线程应用程序在任何框架中都是一种挑战。尽管TPL、PLINQ和委托类型可以在某种程度上简化操作（特别是和其他平台和语言比起来），但开发者仍然需要了解各种高级技术的来龙去脉。

随着.NET 4.5的发布，C#编程语言（以及VB）新增了两个关键字，进一步简化了编写异步代码的过程。相比本章前面介绍的所有示例，使用新的async和await关键字，编译器将使用System.Threading和System.Threading.Tasks命名空间中的众多成员，为我们生成大量线程代码。

19.13.1　C# async和await关键字初探

C# async关键字用来指定某个方法、Lanbda表达式或匿名方法自动以异步的方式来调用。是的，没错。就是用async修饰符声明方法这么简单，CLR会创建新的执行线程来处理任务。在调用async方法时，await关键字会自动暂停当前线程中任何其他活动，直到任务完成，离开调用线程，并继续其幸福之旅。

为了演示这一点，我们创建一个新的Windows Forms应用程序FunWithCSharpAsync，并在窗体的主代码文件（重命名为MainForm）中引入System.Threading命名空间。这之后，将一个Button控件（btnCallMethod）和一个TextBox控件（txtInput）放到设计器表面上，并配置基本的UI属性（颜色、字体、文本）。现在，处理Button控件的Click事件，在事件处理程序中，调用私有辅助方法DoWork()，它强制使线程等待10秒钟。如下所示：

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }

  private void btnCallMethod_Click(object sender, EventArgs e)
  {
    this.Text = DoWork();
  }

  private string DoWork()
  {
    Thread.Sleep(10000);
    return "Done with work!";
  }
}
```

基于本章所学过的知识你会知道，如果运行程序并点击该按钮，需要等待10秒钟，文本框控件才能接收到键盘输入。此外，同样需要等待10秒钟主窗体的标题才会修改为“Done with work!”。

如果要用本章之前介绍的技术使程序响应得更快，需要做很多工作。但在.NET 4.5下，可以编写如下的代码：

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }

  private async void btnCallMethod_Click(object sender, EventArgs e)
  {
    this.Text = await DoWork();
  }

  //对下面的代码进行走查
  private Task<string> DoWork()
  {
    return Task.Run(() =>
    {
```

```
            Thread.Sleep(10000);
            return "Done with work!";
        });
    }
}
```


首先，注意按钮的Click事件处理程序标记了async关键字。这意味着该方法可以作为非阻塞式调用的成员。同样注意该事件处理程序的实现在被调用的方法名前使用了await关键字。这很重要：如果用async关键字修饰某个方法，但方法内部没有一个await方法调用，那么实质上仍将构建一个阻塞的、同步的方法调用（实际上会得到一个编译器警告）。

现在，注意我们需要使用System.Threading.Tasks命名空间中的Task类来重构DoWork()方法使其正常工作。我们不直接返回特定的值（本例中为字符串对象），而是返回一个Task<T>对象，其泛型类型参数T即为实际返回值（能跟上吗？）。

现在DoWork()的实现直接返回Task<T>对象了，它是Task.Run()的返回值。Run()方法接受一个Func<>或Action<>委托。阅读到本书这里的时候你已经知道，我们可以用Lambda表达式来简化委托。基本上，新版的DoWork()在表述下面这样的事实：

新的任务，因为你的调用而运行。旧的任务，从此开始休眠。10秒，只需10秒它就会睡醒，继续运行。它回馈我一串字符，我将把它们放入新的 Task<string>对象，返回给，最初的那个调用者。

将DoWork()的新实现翻译成更自然的语言（像诗一样）之后，我们对await标记的真正作用有了一定的了解。该关键字总是修饰返回Task对象的方法。当逻辑流到达await标记后，调用线程挂起直到调用完成。运行该版本的应用程序，你会发现你可以点击按钮，然后立即在文本区域输入值。10秒钟之后，窗体的标题更新为完整的消息通知。

19.13.2 异步方法的命名约定

现在，新版的DoWork()已经展示，但按钮的Click事件处理程序还如下所示：

```
private async void btnCallMethod_Click(object sender, EventArgs e)
{
  //哦，这里没有await关键字
  this.Text = DoWork();
}
```

注意，我们对该方法使用了async关键字标记，但在调用DoWork()方法时没有用await关键字修饰。这时将会得到编译器错误，因为DoWork()的返回值是一个Task对象，我们将它直接赋值给接受字符串数据类型的Text属性。记住，await标记负责提取包含在Task对象中的内在返回值。由于没有使用该标记，类型就不匹配了。如图19-5所示，编译器将通知我们，要调用的方法是“awaitable”的，并推荐了调用该方法的正确形式。

注意 “awaitable”的方法是指返回Task<T>的方法。



图19-5　返回Task对象的方法可以通过await调用

由于返回Task对象的方法可以通过async和await标记以非阻塞的方式调用，微软推荐（也是最佳实践）任何返回Task的方法都用“Async”作为后缀。这样，了解该命名约定的开发者看到该方法后就会知道，如果要在异步上下文中调用该方法，就需要使用await关键字。

说明　使用async/await关键字的GUI控件的事件处理程序（如按钮的Click处理程序）不遵循这一命名约定（按照约定，应该避免冗余）。

此外，DoWork()方法还可以用async和await标记修饰（尽管当前示例不必严格遵循这一点）。因此，最终修改的示例如下，它符合推荐的命名约定：

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }

  private async void btnCallMethod_Click(object sender, EventArgs e)
  {
    this.Text = await DoWorkAsync();
  }

  private async Task<string> DoWorkAsync()
  {
    return await Task.Run(() =>
    {
      Thread.Sleep(10000);
      return "Done with work!";
    });
  }
}
```

19.13.3 返回void的异步方法

目前，DoWork()方法返回一个Task，它包含调用者可以通过显式的await关键字获取的“真正的数据”。但如何构建一个返回void的匿名方法呢？这时我们要使用非泛型的Task类，并忽略return语句，如下所示：

```
private async Task MethodReturningVoidAsync()
{
  await Task.Run(() => { /*进行一些处理... */
                         Thread.Sleep(4000);
                       });
}
```

该方法的调用者（比如另一个按钮的Click事件处理程序）将按照下面的形式使用await和async关键字：

```
private async void btnVoidMethodCall_Click(object sender, EventArgs e)
{
  await MethodReturningVoidAsync();
  MessageBox.Show("Done!");
}
```

19.13.4 具有多个await的异步方法

单个async方法完全可以在其实现中拥有多个await上下文。假设我们的应用程序存在第三个按钮Click事件处理程序，并且用async关键字标记。在前面的示例中，Click处理程序特意调用了运行真正Task的外部方法。其实你也可以通过一组Lambda表达式来内联这些逻辑，如下所示：

```
private async void btnMutliAwaits_Click(object sender, EventArgs e)
{
  await Task.Run(() => { Thread.Sleep(2000); });
  MessageBox.Show("Done with first task!");

  await Task.Run(() => { Thread.Sleep(2000); });
  MessageBox.Show("Done with second task!");

  await Task.Run(() => { Thread.Sleep(2000); });
  MessageBox.Show("Done with third task!");
}
```

同样，这里的每个任务都只是将当前线程挂起一段时间。但是，任何工作单元都可以用这些任务来表示（如调用Web服务、读取数据库等）。我们将本例中的一些关键点总结如下：

- 方法（包括Lambda表达式和匿名方法）可以用async关键字标记，允许该方法以非阻塞的形式进行工作。
- 用async关键字标记的方法（包括Lambda表达式和匿名方法）在遇到await关键字之前将以阻塞的形式运行。
- 单个async方法可以拥有多个await上下文。
- 当遇到await表达式时，调用线程将挂起，直到await的任务完成。同时，控制将返回给方法的调用者。

- await关键字将从视图中隐藏返回的Task对象，直接返回实际的返回值。没有返回值的方法可以简单地返回void。
- 根据命名约定，要被异步调用的方法应该以“Async”作为后缀。

源代码 FunWithCSharpAysnc项目的源代码位于Chapter 19子目录下。

19.13.5 用async/await改进AddWithThreads示例

在本章开头，我们构建了一个示例AddWithThreads，使用了.NET平台的原始线程API System.Threading。现在，我们来用新的C# async和await关键字改进这一示例，来看看应用程序逻辑可以清晰到何种地步。首先来看看AddWithThreads项目原来是如何工作的。

- 创建自定义的AddParams类，表示要被求和的数据。
- 使用Thread类和ParameterizedThreadStart委托，指向一个接受AddParams对象的Add()方法。
- 使用AutoResetEvent类来确保调用线程会等待次线程完成。

最重要的是，对于在次执行线程计算两个数字之和这种简单的任务来说，所需要的代码太多了。以下是同样的项目，使用学过的.NET 4.5中的新技术进行重构（AddParams类此处不再赘述，只需记得它包含两个字段a和b，来表示数据之和）：

```
class Program
{
  static void Main(string[] args)
  {
    AddAsync();
    Console.ReadLine();
  }

  private static async Task AddAsync()
  {
    Console.WriteLine("***** Adding with Thread objects *****");
    Console.WriteLine("ID of thread in Main(): {0}",
      Thread.CurrentThread.ManagedThreadId);

    AddParams ap = new AddParams(10, 10);
    await Sum(ap);

    Console.WriteLine("Other thread is done!");
  }

  static async Task Sum(object data)
  {
    await Task.Run(() =>
    {
      if (data is AddParams)
      {
        Console.WriteLine("ID of thread in Add(): {0}",
          Thread.CurrentThread.ManagedThreadId);

        AddParams ap = (AddParams)data;
        Console.WriteLine("{0} + {1} is {2}",
```

```
            ap.a, ap.b, ap.a + ap.b);
        }
      });
    }
  }
```


我首先想指出的是，原来在Main()中的代码被移到了新的AddAsync()方法中。原因不仅是符合期望的命名约定，还包括非常重要的一点：

说明 可执行程序中的Main()方法不能用async关键字标记。

注意AddAsync()用async标记，并定义了一个await上下文。同样Sum()方法包含一个新的Task来执行工作单元。不论怎样，运行程序，我们会发现10加10始终是20。注意，我们确实有两个不同的线程ID（如图19-6所示）。

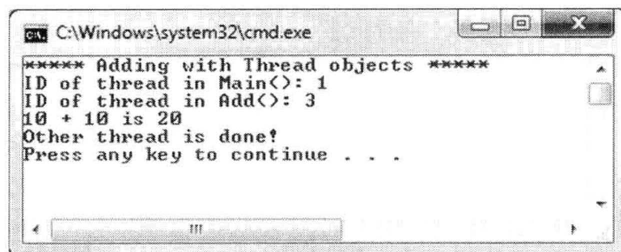


图19-6 用.NET 4.5的异步特性改进的System.Threading示例

源代码 AddWithThreadsAsync项目的源代码位于Charpter 19子目录下。

如你所见，.NET 4.5引入的async和await关键字从根本上简化了再次执行线程调用方法的过程。尽管我们只介绍了几个关于这个C#语言新功能的示例，但你已经为今后的学习打下了一个良好的基础。

19.14 小结

本章首先研究了如何配置.NET委托类型以进行异步调用。读者已经看到了，BeginInvoke()和EndInvoke()方法允许我们使用最小的代价来操作次线程。本章还介绍了IAsyncResult接口和AsyncResult类类型。这些类型提供了多种方式来同步调用线程和获取方法的返回值。

接着分析了System.Threading命名空间的作用。当一个程序创建了多个执行线程后，这个程序的运行结果可能（看起来）在同一时间执行多个任务。我们也研究了保护线程敏感代码的几种方式，确保共享资源不会变成无法使用的脏数据。

本章接着介绍了.NET 4.0引入的关于多线程开发的新模型，如TPL和PLINQ。最后我们介绍了.NET 4.5的C# async和await关键字的作用。如你所见，这两个关键字在后台使用了很多TPL框架的类型，只不过编译器为我们做了大量的工作，来创建复杂的线程和同步代码。

第 20 章

文件输入输出和对象序列化

如果要创建桌面应用程序，那么在用户会话之间保存信息是必不可少的。本章将从.NET Framework的视角来研究一系列与I/O相关的主题。首先研究System.IO命名空间定义的一些重要类型，进而理解怎样以编程方式修改计算机的目录和文件结构。掌握了这些后，接下来的任务就是研究读写基于字符、二进制、字符串、内存的各种数据存储内容的方法。

在学习了如何使用核心I/O类型操作文件和目录之后，我们将介绍对象序列化相关的内容。你可以使用对象序列化将对象的状态持久化为System.IO.Stream的派生类，或从System.IO.Stream派生类中获取对象的状态。当你使用不同的远程技术（如WCF）向远程计算机复制对象时，这种序列化对象的能力就尤为重要。其实，序列化本身也是非常有用的，将在很多.NET应用程序（分布式或非分布式）中发挥作用。

20.1 研究 System.IO 命名空间

在.NET Framework中，System.IO命名空间主要包含基于文件（和基于内存）的输入输出（I/O）服务的相关基础类库。和其他命名空间一样，System.IO定义了一系列类、接口、枚举、结构和委托。它们大多数包含在mscorlib.dll中，另外有一部分System.IO命名空间的成员则包含在System.dll程序集中。注意Visual Studio会自动为项目添加这些程序集的引用。

System.IO命名空间的多数类型主要用于编程操作物理目录和文件，而另一些类型则提供了从字符串缓冲区和内存区域中读写数据的方法。为了让读者了解System.IO功能的概况，表20-1列出了一些主要的（非抽象）类。

表20-1 System.IO命名空间的主要成员

非抽象I/O类类型	作　用
BinaryReader和BinaryWriter	这两个类型能够以二进制值存储和读取基本数据类型（整型、布尔型、字符串型和其他类型）
BufferedStream	这个类型为字节流提供了临时的存储空间，可以以后提交
Directory和DirectoryInfo	这两个类型用来操作计算机的目录结构。Directory类型主要的功能通过静态方法实现。DirectoryInfo类型则通过一个有效的对象引用来实现类似功能
DriveInfo	提供计算机驱动器的详细信息

（续）

非抽象I/O类类型	作　用
File和FileInfo	这两个类型用来操作计算机上的一组文件。File类型主要的功能通过静态成员实现，FileInfo类型则通过一个有效的对象引用来实现类似功能
FileStream	这个类型实现文件随机访问（比如寻址能力），并以字节流来表示数据
FileSystemWatcher	这个类型监控对指定外部文件的更改
MemoryStream	这个类型实现对内存（而不是物理文件）中存储的流数据的随机访问
Path	这个类型对包含文件或目录路径信息的System.String类型执行操作。这些操作是与平台无关的
StreamWriter和StreamReader	这两个类型用来在（从）文件中存储（获取）文本信息。不支持随机文件访问
StringWriter和StringReader	和StreamWriter/StreamReader类型差不多，这两个类型同样和文本信息打交道，不同的是基层的存储器是字符串缓冲区而不是物理文件

除了这些类类型，System.IO还定义了许多枚举类型和一组抽象类（Stream、TextReader和TextWriter等），它们为所有派生类定义了共享的多态接口。本章后面有更多有关这些类型的介绍。

20.2 Directory(Info)和 File(Info)类型

System.IO提供了4个类型来实现对单个文件和计算机目录结构的操作。前两个类型Directory和File通过各种静态成员实现建立、删除、复制和移动操作。与之紧密关联的FileInfo和DirectoryInfo类型则通过实例级方法来实现类似的功能（因此必须用new关键字分配它们）。从图20-1中，我们注意到Directory和File类型直接扩展了System.Object，而DirectoryInfo和FileInfo则从FileSystemInfo抽象类派生。

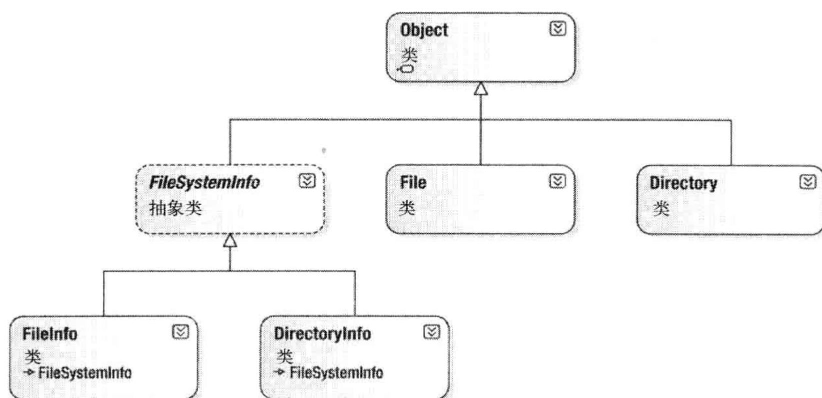


图20-1　File和Directory相关类型

一般说来，FileInfo和DirectoryInfo是获取文件或目录细节（如创建时间、读写能力等）更好的方式，因为它们的成员往往会返回强类型的对象。相反，Directory和File类成员往往会返回简单字符

串值而不是强类型对象。不过，这仅仅是一个准则。在很多情况下，你都可以使用File/FileInfo或Directory/DirectoryInfo完成相同的工作。

FileSystemInfo抽象基类

DirectoryInfo和FileInfo类型实现了许多FileSystemInfo抽象基类的行为。大部分FileSystemInfo类成员的作用是用来获取指定文件或目录的一般特性（比如创建时间、各种特性等）。表20-2列举了一些重要属性。

表20-2 FileSystemInfo属性

属 性	作 用
Attributes	获取或设置与当前文件关联的特性，由FileAttributes枚举表示（例如，是只读、加密、隐藏或压缩文件或目录）
CreationTime	获取或设置当前文件或目录的创建时间
Exists	用来判断指定文件或目录是否存在的值
Extension	获取文件的扩展名
FullName	获取目录或文件的完整路径
LastAccessTime	获取或设置上次访问当前文件或目录的时间
LastWriteTime	获取或设置上次写入当前文件或目录的时间
Name	获取当前文件或目录的名称

FileSystemInfo类型还定义了Delete()方法，该操作由派生类型从硬盘中删除指定文件或目录来实现。同样，在获取文件特性前使用Refresh()方法能确保当前文件（或目录）的统计信息是最新的。

20.3 使用 DirectoryInfo 类型

我们首先要讨论的用于I/O的类型是DirectoryInfo类。它包含一组用来创建、移动、删除和枚举所有目录/子目录的成员。表20-3详细列举了除了它的基类（FileSystemInfo）提供的功能外的一些成员。

表20-3 DirectoryInfo类型的主要成员

成 员	作 用
Create()和CreateSubdirectory()	按照路径名建立一个目录（或者一组子目录）
Delete()	删除一个目录和它的所有内容
GetDirectories()	返回一个表示当前目录中所有子目录的DirectoryInfo对象数组
GetFiles()	返回FileInfo对象数组，表示指定目录下的一组文件
MoveTo()	将一个目录及其内容移动到一个新的路径
Parent	获取指定路径的父目录
Root	获取路径的根部分

我们首先使用DirectoryInfo类型指定一个特别的目录路径作为构造函数的参数。如果需要访问当前应用程序目录的话（比如执行的应用程序的目录），可以使用“.”符号。下面是一些例子：

```
// 绑定到当前的应用程序目录
DirectoryInfo dir1 = new DirectoryInfo(".");

// 使用verbatim字符串绑定到C:\Windows
DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");
```

在第二个例子中，必须确保传入构造函数的路径（C:\Windows）是在物理计算机上存在的。然而如果试图使用一个不存在的目录，系统会引发System.IO.DirectoryNotFoundException异常。因此，如果指定了一个尚未创建的目录的话，在对目录进行操作前首先需要调用Create()方法。

```
// 绑定到一个不存在的目录，然后创建它
DirectoryInfo dir3 = new DirectoryInfo(@"C:\MyCode\Testing");
dir3.Create();
```

创建了DirectoryInfo对象后，就能使用任何一个派生自FileSystemInfo的属性来获取底层目录的内容。为了演示，创建一个新的控制台应用程序DirectoryApp并更新C#文件来导入System.IO。

修改Program类的静态方法，这个方法创建了一个新的DirectoryInfo对象，它映射到C:\Windows（如果需要的话，可以调整路径），显示了许多相关的统计信息：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Directory(Info) *****\n");
    ShowWindowsDirectoryInfo();
    Console.ReadLine();
  }

  static void ShowWindowsDirectoryInfo()
  {
    // 转储目录信息
    DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
    Console.WriteLine("***** Directory Info *****");
    Console.WriteLine("FullName: {0}", dir.FullName);
    Console.WriteLine("Name: {0}", dir.Name);
    Console.WriteLine("Parent: {0}", dir.Parent);
    Console.WriteLine("Creation: {0}", dir.CreationTime);
    Console.WriteLine("Attributes: {0}", dir.Attributes);
    Console.WriteLine("Root: {0}", dir.Root);
    Console.WriteLine("**************************\n");
  }
}
```

你的输出结果可能会不同，但是你应该能看到类似下面的信息：

```
***** Fun with Directory(Info) *****

***** Directory Info *****
FullName: C:\Windows
Name: Windows
Parent:
Creation: 7/13/2012 10:22:32 PM
Attributes: Directory
```

```
Root: C:\
**************************
```

20.3.1 使用DirectoryInfo类型枚举出文件

除了获取已存在目录的基本信息外，我们还能使用DirectoryInfo类型的一些方法来扩展当前的例子。首先，使用GetFiles()方法来获取C:\Windows\Web\Wallpaper目录下所有*.jpg文件的信息。

说明 如果我们的机器没有C:\Windows\Web\Wallpaper目录，则修改代码来读取机器目录里的文件（例如从C:\Windows目录读取所有的*.bmp文件）。

GetFiles()方法返回FileInfo类型的数组，每个FileInfo类型都包含了一个文件的细节（有关FileInfo类型的详细内容在本章后面进行讨论）。假定从Main()中调用了Program类的静态方法：

```
static void DisplayImageFiles()
{
  DirectoryInfo dir = new DirectoryInfo(@"C:\Windows\Web\Wallpaper");
  // 获取所有*.jpg文件
  FileInfo[] imageFiles = dir.GetFiles("*.jpg", SearchOption.AllDirectories);

  // 我们找到多少文件
  Console.WriteLine("Found {0} *.jpg files\n", imageFiles.Length);

  // 输出每个文件的信息
  foreach (FileInfo f in imageFiles)
  {
    Console.WriteLine("***************************");
    Console.WriteLine("File name: {0}", f.Name);
    Console.WriteLine("File size: {0}", f.Length);
    Console.WriteLine("Creation: {0}", f.CreationTime);
    Console.WriteLine("Attributes: {0}", f.Attributes);
    Console.WriteLine("***************************\n");
  }
}
```

注意，在调用GetFiles()时指定了搜索选项（search option）。这样就可以查看所有子目录。运行该应用程序，将会列出所有匹配搜索模式的文件。

20.3.2 使用DirectoryInfo类型创建子目录

我们能使用DirectoryInfo.CreateSubdirectory()方法以编程方式扩展目录结构。使用这个方法，可以建立一个子目录，也可以一次建立多个嵌套子目录。例如，下面这段代码通过建立一些自定义子目录来扩展C:的目录结构：

```
static void ModifyAppDirectory()
{
  DirectoryInfo dir = new DirectoryInfo(@"C:\");

  // 在应用程序目录下创建\MyFolder
  dir.CreateSubdirectory("MyFolder");
```

```
    // 在应用程序目录下创建\MyFolder2\Data
    dir.CreateSubdirectory(@"MyFolder2\Data");
}
```

如果从Main()调用这个方法并使用Windows资源管理器来检查Windows目录，会发现这些子目录已被成功创建（如图20-2所示）。

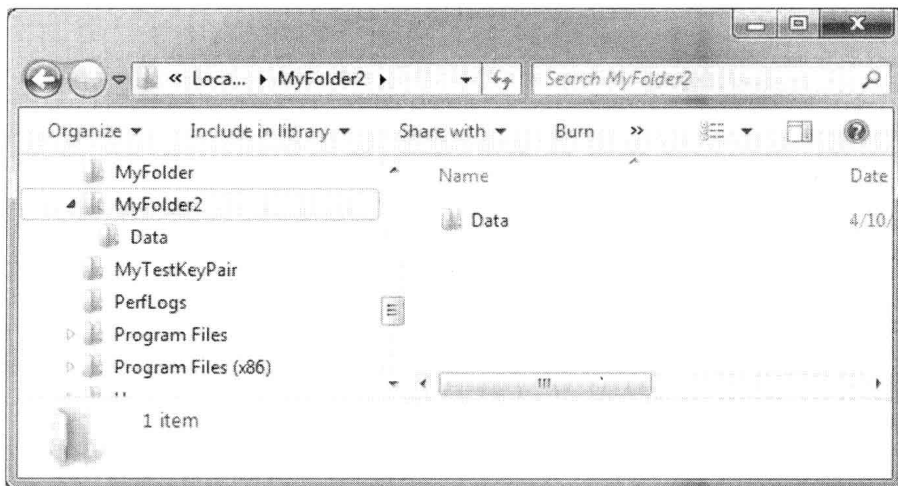


图20-2　创建子目录

尽管不一定要去捕获CreateSubdirectory()方法的返回值，但是需要知道的是，如果执行成功，它会返回表示新建项的DirectoryInfo类型。考虑对前面方法的修改。注意，DirectoryInfo构造函数中的点标记，它允许我们访问应用程序的安装点。

```
static void ModifyAppDirectory()
{
  DirectoryInfo dir = new DirectoryInfo(".");

  // 在初始目录下创建\MyFolder
  dir.CreateSubdirectory("MyFolder");

  // 捕获返回的DirectoryInfo类型
  DirectoryInfo myDataFolder = dir.CreateSubdirectory(@"MyFolder2\Data");

  // 在..\MyFolder2\Data上输出路径
  Console.WriteLine("New Folder is: {0}", myDataFolder);
}
```

20.4　使用 Directory 类型

在实践过了DirectoryInfo以后，我们来研究Directory类型。Directory的静态成员实现了由DirectoryInfo定义的实例级成员的大部分功能。前面说过，Directory成员返回的是字符串数据而不是强类型的FileInfo和DirectoryInfo对象。

为了演示Directory类型的一些功能，最后一个辅助函数显示了所有映射到当前计算机的驱动器（通过Directory.GetLogicalDrivers()），然后使用Directory.Delete()静态方法移除前面建立的\MyFolder和\MyFolder2\Data子目录：

```
static void FunWithDirectoryType()
{
  // 列出当前电脑的所有驱动器
  string[] drives = Directory.GetLogicalDrives();
  Console.WriteLine("Here are your drives:");
  foreach (string s in drives)
    Console.WriteLine("--> {0} ", s);

  // 删除前面建立的目录
  Console.WriteLine("Press Enter to delete directories");
  Console.ReadLine();
  try
  {
    Directory.Delete(@"C:\MyFolder");

    // 第二个参数指定你是否希望删除某个子目录
    Directory.Delete(@"C:\MyFolder2", true);
  }
  catch (IOException e)
  {
    Console.WriteLine(e.Message);
  }
}
```

源代码　DirectoryApp项目的源代码位于Chapter 20子目录下。

20.5　使用DriveInfo类类型

System.IO命名空间提供了一个叫做DriveInfo的类。和Directory.GetLogicalDrives()相似，DriveInfo.GetDrives()静态方法能获取计算机上驱动器的名字。然而和Directory.GetLogicalDrives()不同，DriveInfo提供了许多其他的细节（比如驱动器类型、可用空间、卷标等）。请看下面定义在新的控制台应用程序DriveInfoApp中的Program类（不要忘记导入System.IO）：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with DriveInfo *****\n");

    // 得到所有驱动器的信息
    DriveInfo[] myDrives = DriveInfo.GetDrives();
    // 输出驱动器统计信息
    foreach(DriveInfo d in myDrives)
    {
      Console.WriteLine("Name: {0}", d.Name);
      Console.WriteLine("Type: {0}", d.DriveType);

      // 检查驱动器是否已经准备好
```

```
        if(d.IsReady)
        {
          Console.WriteLine("Free space: {0}", d.TotalFreeSpace);
          Console.WriteLine("Format: {0}", d.DriveFormat);
          Console.WriteLine("Label: {0}", d.VolumeLabel);
  }
         Console.WriteLine();

      }
      Console.ReadLine();
    }
  }
```

下面显示了可能的输出结果：

```
***** Fun with DriveInfo *****

Name: C:\
Type: Fixed
Free space: 587376394240
Format: NTFS
Label: Mongo Drive

Name: D:\
Type: CDRom
Name: E:\
Type: CDRom
Name: F:\
Type: CDRom
Name: H:\
Type: Fixed
Free space: 477467508736
Format: FAT32
Label: My Passport
```

至此，我们已经研究了Directory、DirectoryInfo和DriveInfo类的一些重要的行为属性。下一步学习如何创建、打开、关闭和删除填充到指定目录中的文件。

源代码 DriveInfoApp项目的源代码位于Chapter 20子目录下。

20.6 使用 FileInfo 类

从DirectoryApp例子中可以看到，FileInfo类能让我们获得硬盘上现有文件的详细信息(创建时间、大小、文件特性等)，并帮助我们创建、复制、移动和删除文件。除了从FileSystemInfo继承的一些功能外，表20-4列出了一些FileInfo类独有的核心成员。

表20-4　FileInfo核心成员

成　　员	作　　用
AppendText()	创建一个StreamWriter类型（后面会讨论），它用来向文件追加文本
CopyTo()	将现有文件复制到新文件
Create()	创建一个新文件并且返回一个FileStream类型（后面会讨论），通过它来和新创建的文件进行交互
CreateText()	创建一个写入新文本文件的StreamWriter对象
Delete()	删除FileInfo实例绑定的文件
Directory	获取父目录的实例
DirectoryName	获取父目录的完整路径
Length	获取当前文件的大小
MoveTo()	将指定文件移到新位置，并提供指定新文件名的选项
Name	获取文件名
Open()	用各种读/写访问权限和共享特权打开文件
OpenRead()	创建只读FileStream对象
OpenText()	创建从现有文本文件中读取数据的StreamReader（后面会讨论）
OpenWrite()	创建只写FileStream类型

注意，大部分FileInfo类的成员返回一个I/O相关的特定对象（FileStream和StreamWriter等），让我们以不同格式从关联文件读或向关联文件写数据。下面会谈到这些类型，不过现在先来研究使用FileInfo类类型来获取一个文件句柄的各种方法。

20.6.1　FileInfo.Create()方法

第一种建立文件句柄的方法是使用FileInfo.Create()方法：

```
static void Main(string[] args)
{
  // 在C盘新建一个文件
  FileInfo f = new FileInfo(@"C:\Test.dat");
  FileStream fs = f.Create();

  // 使用FileStream对象

  // 关闭文件流
  fs.Close();
}
```

需要注意的是，FileInfo.Create()方法返回一个FileStream对象，FileStream能对基层的文件进行同步/异步的读/写操作。需要知道的是，FileInfo.Create()返回的FileStream对象给所有的用户授予完全读写操作权限。

还要注意，在使用了当前FileStream对象之后，要确保关闭句柄来释放流的底层非托管资源。由于FileStream实现了IDisposable，所以我们可以使用C#的using域来让编译器生成释放逻辑（详见第8章）。

```
static void Main(string[] args)
{
  // 最好为I/O类型定义using域
  FileInfo f = new FileInfo(@"C:\Test.dat");
  using (FileStream fs = f.Create())
  {
    // 使用FileStream对象
  }
}
```


20.6.2 FileInfo.Open()方法

我们能使用FileInfo.Open()方法来打开现有文件，同时也能使用它来创建新文件，它比FileInfo.Create()多了很多细节，因为Open()通常有好几个参数，可以限定所操作的文件的整体结构。一旦调用Open()完成后，它返回一个FileStream对象，请看下面代码：

```
static void Main(string[] args)
{
  // 通过FileInfo.Open()创建新文件
  FileInfo f2 = new FileInfo(@"C:\Test2.dat");
  using(FileStream fs2 = f2.Open(FileMode.OpenOrCreate,
    FileAccess.ReadWrite, FileShare.None))
  {
    // 使用FileStream对象
  }
}
```

上面的重载Open()方法需要3个参数。第一个参数指定I/O请求的基本方式（比如说新建文件、打开现有文件和追加文件等），它的值由FileMode枚举指定（详情参见表20-5）：

```
public enum FileMode
{
  CreateNew,
  Create,
  Open,
  OpenOrCreate,
  Truncate,
  Append
}
```

表20-5 FileMode枚举的成员

成员	作用
CreateNew	通知操作系统新建文件。如果存在，就会抛出IOException
Create	通知操作系统新建文件。如果存在，就会被覆盖
Open	打开既有文件。如果文件不存在，就会抛出FileNotFoundException
OpenOrCreate	如果文件存在，则打开，否则新建文件
Truncate	打开文件并截断文件为0字节大小
Append	打开文件，移动到文件尾部，开始写操作（这个标识只能和只写流一起使用）。如果文件不存在，则新建文件

第二个参数的值由FileAccess枚举定义，用来决定基层流的读写行为：

```
public enum FileAccess
{
  Read,
  Write,
  ReadWrite
}
```

最后，第三个参数FileShare指定文件在其他文件处理程序中的共享方式。下面是一些主要成员：

```
public enum FileShare
{
  Delete,
  Inheritable,
  None,
  Read,
  ReadWrite,
  Write
}
```

20.6.3 FileInfo.OpenRead()和 FileInfo.OpenWrite()方法

FileInfo.Open()方法能让我们用非常灵活的方式获取文件句柄，FileInfo类同样提供了OpenRead()和OpenWrite()成员。读者可能也想到了，这些方法不需要提供各种枚举值，就能返回一个正确配置的只读或只写的FileStream类型。和FileInfo.Create()、FileInfo.Open()方法一样，OpenRead()和OpenWrite()也都返回一个FileStream对象（假定你的C盘上有名为Test3.dat和Test4.dat的文件）：

```
static void Main(string[] args)
{
  // 得到一个只读的FileStream对象
  FileInfo f3 = new FileInfo(@"C:\Test3.dat");
  using(FileStream readOnlyStream = f3.OpenRead())
  {
    // 使用FileStream对象
  }

  // 得到一个只写的FileStream对象
  FileInfo f4 = new FileInfo(@"C:\Test4.dat");
  using(FileStream writeOnlyStream = f4.OpenWrite())
  {
    // 使用FileStream对象……
  }
}
```

20.6.4 FileInfo.OpenText()方法

FileInfo类型另外一个“Open”成员是OpenText()。和Create()、Open()、OpenRead()、OpenWrite()方法不同，OpenText()方法返回的是一个StreamReader类型（而不是FileStream类型）的实例。假定你的C盘中有名为boot.ini的文件，可以用以下方法访问其中的内容：

```
static void Main(string[] args)
{
  // 得到一个StreamReader对象
  FileInfo f5 = new FileInfo(@"C:\boot.ini");
```

```
    using(StreamReader sreader = f5.OpenText())
    {
      // 使用StreamReader对象……
    }
  }
```

马上就会看到，StreamReader类型提供了从基层文件读取字符数据的方法。

20.6.5 FileInfo.CreateText()和FileInfo.AppendText()方法

最后需要指出的两个方法是CreateText()和AppendText()，它们都返回一个StreamWriter对象，代码如下：

```
static void Main(string[] args)
{
  FileInfo f6 = new FileInfo(@"C:\Test6.txt");
  using(StreamWriter swriter = f6.CreateText())
  {
    // 使用StreamWriter对象
  }

  FileInfo f7 = new FileInfo(@"C:\FinalTest.txt");
  using(StreamWriter swriterAppend = f7.AppendText())
  {
    // 使用StreamWriter对象
  }
}
```

读者可能猜到了，StreamWriter类型提供向基层文件写入字符数据的方法。

20.7 使用 File 类型

File类型的静态成员提供了和FileInfo类型差不多的功能。与FileInfo类似，File类提供了AppendText()、Create()、CreateText()、Open()、OpenRead()、OpenWrite()和OpenText()方法。其实，在大多数情况下，File和FileInfo类型能互换使用。例如，前面每一个FileStream示例都可以用File类型来简化：

```
static void Main(string[] args)
{
  // 通过File.Create()获取FileStream对象
  using(FileStream fs = File.Create(@"C:\Test.dat"))
  {}

  // 通过File.Open()获取FileStream对象
  using(FileStream fs2 = File.Open(@"C:\Test2.dat",
    FileMode.OpenOrCreate,
    FileAccess.ReadWrite, FileShare.None))
  {}

  // 得到一个只读权限的FileStream对象
  using(FileStream readOnlyStream = File.OpenRead(@"Test3.dat"))
  {}

  // 得到一个只写权限的FileStream对象
  using(FileStream writeOnlyStream = File.OpenWrite(@"Test4.dat"))
  {}
```

```
    // 得到一个StreamReader对象
    using(StreamReader sreader = File.OpenText(@"C:\boot.ini"))
    {}

    // 得到一些StreamWriter对象
    using(StreamWriter swriter = File.CreateText(@"C:\Test6.txt"))
    {}

    using(StreamWriter swriterAppend = File.AppendText(@"C:\FinalTest.txt"))
    {}
  }
```

其他File成员

File类型提供了一些独有的成员，表20-6列举了其中的一些成员，它们可以极大地简化读写文本数据的过程。

表20-6　File类型的方法

方　法	作　用
ReadAllBytes()	打开指定文件，以字节数组形式返回二进制数据，然后关闭文件
ReadAllLines()	打开指定文件，以字符串数组形式返回字符数据，然后关闭文件
ReadAllText()	打开指定文件，以System.String形式返回字符数据，然后关闭文件
WriteAllBytes()	打开指定文件，写入字节数组，然后关闭文件
WriteAllLines()	打开指定文件，写入字符串数组，然后关闭文件
WriteAllText()	打开指定文件，写入字符数据，然后关闭文件

使用File类型的这些新方法，只用几行代码就可以批量读写数据。更好的是，每一个成员都自动关闭基层文件句柄。例如，下面的控制台程序SimpleFileIO能轻松地将字符串数据保存到C盘的新文件里，并将其读入内存中（本例假设已经导入System.IO）：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Simple I/O with the File Type *****\n");
    string[] myTasks = {
      "Fix bathroom sink", "Call Dave",
      "Call Mom and Dad", "Play Xbox 360"};

    // 向C盘的文件写入所有数据
    File.WriteAllLines(@"C:\tasks.txt", myTasks);

    // 重新读取然后输出
    foreach (string task in File.ReadAllLines(@"C:\tasks.txt"))
    {
      Console.WriteLine("TODO: {0}", task);
    }
    Console.ReadLine();
  }
}
```

很明显，如果你希望快速获取文件句柄的话，使用File类型能节省很多代码。然而使用前面提到的FileInfo对象的好处是能从FileSystemInfo抽象基类定义的成员中获取文件属性。

源代码 SimpleFileIO项目的源代码位于Chapter 20子目录下。

20.8 Stream 抽象类

至此，我们知道了许多获取FileStream、StreamReader和StreamWriter对象的方法，但是还没有使用这些对象从文件读取数据或者向文件写入数据。为了理解怎样去做，首先需要了解“流”的概念。在I/O操作中，流代表了在源文件和目标文件之间传输的一定量的数据。无论使用什么设备（文件、网络连接和打印机等）存储或者显示字节，“流”都能提供一种通用的方式来和字节队列进行交互。

抽象类System.IO.Stream定义了许多成员来提供对存储媒介（比如基层的文件或内存地址）实现同步或异步交互的支持。

说明 “流”的概念不仅仅局限于文件输入/输出。.NET类库提供了“流”来访问网络、内存地址和其他一些与流相关的抽象设备。

Stream派生类型把数据表现为原始的字节流，因此，使用原始的Stream类型有点模糊。一些从Stream派生的类型支持寻址（指获取和调整当前在流中位置的过程）。要理解Stream类提供的功能，可以先看看表20-7列举的一些主要成员。

表20-7 抽象Stream成员

成 员	作 用
CanRead、CanWrite和CanSeek	检测当前流是否支持读、寻址和写
Close()	关闭当前流并释放与之关联的所有资源（如套接字和文件句柄）。在内部，这个方法是Dispose()方法的别名，因此“关闭流”从功能上说等价于“释放流”
Flush()	使用当前的缓冲状态更新基层的数据源或储存库。如果流不实现缓冲，这个方法什么都不做
Length	返回流的长度，以字节来表示
Position	检测在当前流中的位置
Read()和ReadByte()	从当前流读取字节序列（或一个字节），并将此流中的位置偏移读取的字节数
Seek()	设置当前流中的位置
SetLength()	设置当前流的长度
Write()和WriteByte()	向当前流中写入字节序列（或一个字节），并将此流中的当前位置偏移写入的字节数

使用FileStream

FileStream类以合适的方式为基于文件的流提供了抽象Stream成员的实现。这是一个相当原始的流，它只能读取或写入一个字节或者字节数组。其实，我们通常不需要直接和FileStream类型的成员交互，而是使用各种Stream包装类，它们能更方便地处理文本数据和.NET类型。为了说明，让我们体验一下FileStream类型的同步读写能力。

新建一个名为FileStreamApp的控制台应用程序，并将System.IO和System.Text导入到初始的C#代码文件中。目标是把一段简单的文字信息写入一个新建的文件myMessage.dat。然而，因为FileStream只能处理原始字节，我们必须把System.String编码成相应的字节数组。幸好System.Text命名空间定义了一个Encoding类型，它提供了一些成员，来实现在字符串和字节数组之间的编码/解码操作（请查看.NET Framework 4.5 SDK文档，了解Encoding类型的完整细节）。

编码完成后，使用FileStream.Write()方法把字节数组保存到文件内。如果要把这些字节读回内存，还需要（通过Position属性）重置流内部的位置，然后调用ReadByte()方法。最后，可以在控制台上显示这些原始字节或者解码过的字符串。下面是完整的Main()方法：

```
// 不要忘记导入System.Text和System.IO命名空间
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with FileStreams *****\n");

  // 获取一个FileStream对象
  using(FileStream fStream = File.Open(@"C:\myMessage.dat",
    FileMode.Create))
  {
    // 把字符串编码成字节数组
    string msg = "Hello!";
    byte[] msgAsByteArray = Encoding.Default.GetBytes(msg);

    // 把byte[]写入文件
    fStream.Write(msgAsByteArray, 0, msgAsByteArray.Length);

    // 重置流内部的位置
    fStream.Position = 0;

    // 从文件读取字节并显示在控制台
    Console.Write("Your message as an array of bytes: ");
    byte[] bytesFromFile = new byte[msgAsByteArray.Length];
    for (int i = 0; i < msgAsByteArray.Length; i++)
    {
      bytesFromFile[i] = (byte)fStream.ReadByte();
      Console.Write(bytesFromFile[i]);
    }

    // 显示解码后的字符串
    Console.Write("\nDecoded Message: ");
    Console.WriteLine(Encoding.Default.GetString(bytesFromFile));
  }
  Console.ReadLine();
}
```

然而，这个例子在演示把数据填充进文件的同时，也体现出了直接使用FileStream类型的缺点：

需要操作原始字节。其他从Stream派生的类型也差不多。比如，如果想向一段内存写入字节队列的话，可以使用MemoryStream。同样，如果想向网络连接压入字节数组的话，可以使用NetworkStream类型（在System.Net.Sockets命名空间中）。

前面提到过，System.IO命名空间提供了一些读取器和编写器类型来封装从Stream派生的类型的一些细节。

源代码　FileStreamApp项目的源代码位于Chapter 20子目录下。

20.9 使用 StreamWriter 和 StreamReader 类型

当需要读写基于字符的数据（比如字符串）的时候，StreamWriter和StreamReader类就非常有用了。它们都默认使用Unicode字符，当然我们也可以提供一个正确配置的System.Text.Encoding对象引用来改变默认配置。为了使例子更简单，假设默认的Unicode编码能满足我们的需求。

StreamReader和相关的StringReader类型（本章后面会讨论）一样，它们都从TextReader抽象类型派生。TextReader基类为这些派生类型提供了一套非常有限的功能，特别是读取字符流。

StreamWriter类型（和后面会讨论的StringWriter一样）从TextWriter抽象基类派生。这个类定义了一些成员，使得派生的类型能向某个字符流写入文本数据。

为了帮助读者理解StreamWriter和StringWriter类主要的功能，表20-8列举了一些TextWriter抽象基类的核心成员。

表20-8　TextWriter核心成员

成　员	作　用
Close()	关闭当前编写器并释放任何与该编写器关联的系统资源。在这个过程中，缓冲区将被自动清理（这个成员在功能上等同于调用Dispose()方法）
Flush()	清理当前编写器的所有缓冲区，使所有缓冲数据写入基础设备，但是不关闭编写器
NewLine	代表派生的编写器类的行结束符字符串。默认行结束符字符串是回车符后接一个换行符（\r\n）
Write()	这个重载的方法将一行写入文本流，不跟行结束符
WriteLine()	这个重载的方法将一行写入文本流，后跟行结束符

说明　TextWriter类的最后两个成员可能对读者来说很熟悉。前面说过，System.Console类型就有Write()和WriteLine()成员来向标准输出设备写入文本数据。其实，Console.In属性包装了一个TextWriter，Console.Out属性包装了一个TextReader。

派生的StreamWriter类提供了对Write()、Close()和Flush()方法的有效实现，而且还定义了AutoFlush属性。如果把这个属性设置为true的话，StreamWriter会在每次执行一个写操作后，立即写入数据并清理缓冲区。设置AutoFlush为false能获得更好的性能，这样的话，使用StreamWriter完成写操作后需要调用Close()。

20.9.1　写文本文件

现在举一个使用StreamWriter类型的例子，创建一个新的控制台应用程序StreamWriterReaderApp并且导入System.IO命名空间。下面的Main()方法使用File.CreateText()方法新建一个reminders.txt文件。使用返回的StreamWriter对象向新建的文件增加一些文本数据，代码如下：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");

  // 得到一个StreamWriter对象并写入字符串数据
  using(StreamWriter writer = File.CreateText("reminders.txt"))
  {
    writer.WriteLine("Don't forget Mother's Day this year...");
    writer.WriteLine("Don't forget Father's Day this year...");
    writer.WriteLine("Don't forget these numbers:");
    for(int i = 0; i < 10; i++)
      writer.Write(i + " ");

    // 插入一个新行
    writer.Write(writer.NewLine);
  }

  Console.WriteLine("Created file and wrote some thoughts...");
  Console.ReadLine();
}
```

运行程序来检查新建文件的内容（如图20-3所示）。因为在调用CreateText()时没有指定完整路径，所以将会在当前应用程序的bin\Debug文件夹下找到这个新文件。

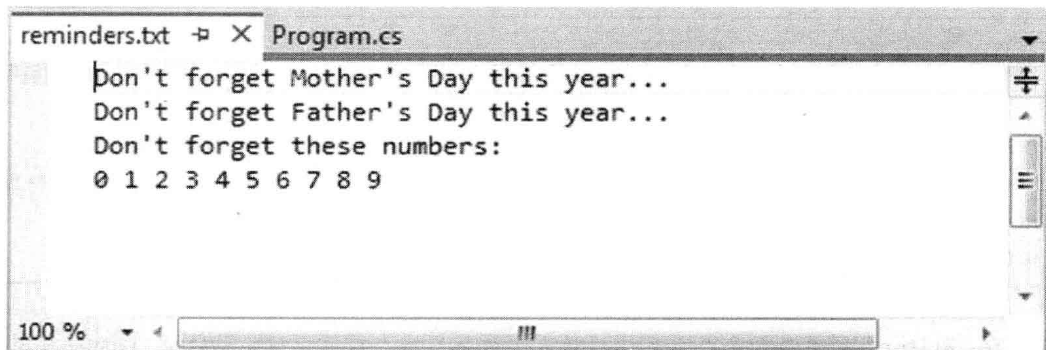


图20-3　*.txt文件的内容

20.9.2　读文本文件

现在需要理解怎样使用相应的StreamReader类型通过编程从文件读取数据。前面说过，这个类从TextReader派生，表20-9列举了一些它的功能。

表20-9 TextReader主要成员

成 员	作 用
Peek()	返回下一个可用字符，而不更改读取器位置。返回-1表示已经到了流的尾部
Read()	从输入流中读取数据
ReadBlock()	从当前流中读取最大计数字符，并从索引开始将该数据写入缓冲区
ReadLine()	从当前流中读取一行字符，并将数据作为字符串返回（返回空字符串代表EOF）
ReadToEnd()	读取从当前位置到流结尾的所有字符，并将它们作为一个字符串返回

如果扩展当前的MyStreamWriterReader类来使用StreamReader，可以从reminders.txt读取文本数据，代码如下：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");
...
  // 现在开始从文件读数据
  Console.WriteLine("Here are your thoughts:\n");
  using(StreamReader sr = File.OpenText("reminders.txt"))
  {
    string input = null;
    while ((input = sr.ReadLine()) != null)
    {
      Console.WriteLine (input);
    }
  }
  Console.ReadLine();
}
```

运行程序后，会发现reminders.txt里的字符数据显示到了控制台。

20.9.3 直接创建StreamWriter/StreamReader类型

可能读者还有一点困惑，那就是使用System.IO的这些类型可以有很多种方法实现相同的结果。比如，我们可以使用File或者FileInfo类型的CreateText()方法来获取StreamWriter。其实，还有一个方法来使用StreamWriter和StreamReader：直接创建它们。例如，现在的应用程序可以进行如下修改：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StreamWriter / StreamReader *****\n");

  // 得到一个StreamWriter，然后写字符串数据
  using(StreamWriter writer = new StreamWriter("reminders.txt"))
  {
    ...
  }

  // 从文件读取数据
  using(StreamReader sr = new StreamReader("reminders.txt"))
  {
    ...
  }
}
```

使用这么多看上去差不多的方法实现文件I/O操作虽说让我们有点困惑，但是这样的确增加了很多灵活性。我们已经看到了怎样使用StreamWriter和StreamReader类从指定文件写入、读取信息。下面再来研究StringWriter和StringReader类型的作用。

源代码　StreamWriterReaderApp项目的源代码位于Chapter 20子目录下。

20.10　使用 StringWriter 和 StringReader 类型

使用StringWriter和StringReader类型，我们可以将文本信息当做内存中的字符一样来处理。当想为基层缓冲区添加基于字符的信息的时候，它们就非常有用。在下面的控制台应用程序StringReaderWriterApp中，我们向一个StringWriter对象（而不是在本地硬盘上的一个文件）写入一段字符串信息：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with StringWriter / StringReader *****\n");

  // 创建一个StringWriter并把字符数据写入内存
  using(StringWriter strWriter = new StringWriter())
  {
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    // 获取内容副本（存储在字符串中）并向控制台输出
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);
  }
  Console.ReadLine();
}
```

因为StringWriter和StreamWriter都从一个基类（TextWriter）派生，它们的写操作逻辑代码或多或少有点相同。但需要知道，StringWriter还有一个特点，那就是它能通过GetStringBuilder()方法来获取一个System.Text.StringBuilder对象：

```
using (StringWriter strWriter = new StringWriter())
{
  strWriter.WriteLine("Don't forget Mother's Day this year...");
  Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

  // 得到内部的StringBuilder
  StringBuilder sb = strWriter.GetStringBuilder();
  sb.Insert(0, "Hey!! ");
  Console.WriteLine("-> {0}", sb.ToString());
  sb.Remove(0, "Hey!! ".Length);
  Console.WriteLine("-> {0}", sb.ToString());
}
```

可使用相应的StringReader类型从字符数据流中读取信息，可以看到，实现方法和相关的StreamReader类型差不多。其实，StringReader类型只不过是通过重写派生的成员来从一段字符数据而不是从一个文件中读取信息，代码如下：

```
using (StringWriter strWriter = new StringWriter())
{
```

```
    strWriter.WriteLine("Don't forget Mother's Day this year...");
    Console.WriteLine("Contents of StringWriter:\n{0}", strWriter);

    // 从StringWriter读取数据
    using (StringReader strReader = new StringReader(strWriter.ToString()))
    {
      string input = null;
      while ((input = strReader.ReadLine()) != null)
      {
        Console.WriteLine(input);
      }
    }
  }
```

源代码 StringReaderWriterApp项目的源代码位于Chapter 20子目录下。

20.11 使用 BinaryWriter 和 BinaryReader

最后研究的读取器/编写器组是BinaryReader和BinaryWriter，它们都从System.Object直接派生。这些类型可以让我们从基层流中以简洁的二进制格式读取或写入离散数据类型。BinaryWriter类型定义了一个多次重载的Write()方法，用于把数据类型写入基层的流。除了Write()方法，BinaryWriter还提供了另外一些成员让我们能获取或设置从Stream派生的类型，并且提供了随机数据访问的支持（如表20-10所示）。

表20-10 BinaryWriter核心成员

成　员	作　用
BaseStream	这个只读属性提供了BinaryWriter对象使用的基层流的访问
Close()	这个方法关闭二进制流
Flush()	这个方法刷新二进制流
Seek()	这个方法设置当前流的位置
Write()	这个方法将值写入当前流

BinaryReader类补充了BinaryWriter的功能，表20-11列出了其中的一些成员。

表20-11 BinaryReader核心成员

成　员	作　用
BaseStream	这个只读属性提供了对BinaryReader对象使用的基层流的访问
Close()	这个方法关闭二进制阅读器
PeekChar()	这个方法返回下一个可用的字符，并且不改变指向当前字节或字符的指针位置
Read()	读取给定的字节或字符，并把它们存入数组
ReadXXXX()	BinaryReader类定义了许多Read()方法来从流中获取下一个类型（ReadBoolean()、ReadByte()和ReadInt32()等）

下面的控制台应用程序BinaryWriterReader新建了一个*.dat文件并且写入了一些数据类型：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Binary Writers / Readers *****\n");

  // 为文件打开一个二进制编写器
  FileInfo f = new FileInfo("BinFile.dat");
  using(BinaryWriter bw = new BinaryWriter(f.OpenWrite()))
  {
    // 输出BaseStream的类型（这里是System.IO.FileStream）
    Console.WriteLine("Base stream is: {0}", bw.BaseStream);

    // 在文件中存储一些数据
    double aDouble = 1234.67;
    int anInt = 34567;
    string aString = "A, B, C";

    // 写数据
    bw.Write(aDouble);
    bw.Write(anInt);
    bw.Write(aString);
  }
  Console.WriteLine("Done!");
  Console.ReadLine();
}
```

注意，从FileInfo.OpenWrite()返回的FileStream对象被传到BinaryWriter类型的构造函数中。使用这项技术，就能很方便地在写入数据前引入一个流。需要理解的是，BinaryWriter构造函数能接受任何派生自Stream类型的参数（比如FileStream、MemoryStream或BufferedStream）。因此，如果希望向内存中写二进制信息的话，只需提供一个有效的MemoryStream对象即可。

BinaryReader类型提供了很多选项来从BinFile.dat文件中读取数据。在这里，我们通过调用与读相关的成员来检测流中的数据：

```
static void Main(string[] args)
{
...
  FileInfo f = new FileInfo("BinFile.dat");
...
  // 从流中读取二进制数据
  using(BinaryReader br = new BinaryReader(f.OpenRead()))
  {
    Console.WriteLine(br.ReadDouble());
    Console.WriteLine(br.ReadInt32());
    Console.WriteLine(br.ReadString());
  }
  Console.ReadLine();
}
```

源代码　BinaryWriterReader项目的源代码位于Chapter 20子目录下。

20.12　以编程方式“观察”文件

现在知道了各种读取器和编写器的用法，下面看看FileSystemWatcher类的作用。当我们想通过编

程监控（或者观察）系统上的文件的时候，这个类就非常有用。特别是，我们能通过System.IO.Notify-Filters枚举来决定需要FileSystemWatcher类型监控文件的行为（大多数成员都能通过字面了解其含义，详细信息请参考.NET Framework 4.5 SDK文档）：

```
public enum NotifyFilters
{
  Attributes, CreationTime,
  DirectoryName, FileName,
  LastAccess, LastWrite,
  Security, Size
}
```

使用FileSystemWatcher类型的第一步是设置Path属性，以指定需要监控的文件所在文件夹的名字（或者位置），还有就是定义需要监控文件的扩展名的Filter属性。

至此，我们可以选择使用FileSystemEventHandler委托关联的事件来实现Changed、Created和Deleted事件的处理方法。这个委托可以调用任何符合下列模式的方法：

```
// FileSystemEventHandler委托必须指向符合下列签名的方法
void MyNotificationHandler(object source, FileSystemEventArgs e)
```

同样，Renamed事件也可以通过RenamedEventHandler委托类型来处理，调用的方法必须符合下列签名：

```
// RenamedEventHandler委托必须指向符合下列签名的方法
void MyRenamedHandler(object source, RenamedEventArgs e)
```

尽管可以使用传统的委托和事件语法来处理各个事件，但我们还可以使用Lambda表达式语法（本项目的可下载代码就使用了Lambda语法，感兴趣的话可以研究研究）。

接下来我们来看看观察文件的过程，假设已经在C盘新建了一个名为MyFolder的目录，其中包含了各种*.txt文件（什么名字都可以）。下面这个控制台应用程序MyDirectoryWatcher将会监控这些MyFolder下的*.txt文件，并显示出文件的建立、删除、修改和重命名事件的消息：

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Amazing File Watcher App *****\n");

  // 确定指向要观察的目录的路径
  FileSystemWatcher watcher = new FileSystemWatcher();
  try
  {
    watcher.Path = @"C:\MyFolder";
  }
  catch(ArgumentException ex)
  {
    Console.WriteLine(ex.Message);
    return;
  }
  // 设置需要“留意”的事情
  watcher.NotifyFilter = NotifyFilters.LastAccess
    | NotifyFilters.LastWrite
    | NotifyFilters.FileName
    | NotifyFilters.DirectoryName;

  // 只观察文本文件
  watcher.Filter = "*.txt";
```

```
    // 增加事件处理程序
    watcher.Changed += new FileSystemEventHandler(OnChanged);
    watcher.Created += new FileSystemEventHandler(OnChanged);
    watcher.Deleted += new FileSystemEventHandler(OnChanged);
    watcher.Renamed += new RenamedEventHandler(OnRenamed);

    // 开始观察目录
    watcher.EnableRaisingEvents = true;

    // 等待用户退出程序
    Console.WriteLine(@"Press 'q' to quit app.");
    while(Console.Read()!='q');
  }
```

这两个事件处理程序输出了当前文件的修改：

```
  static void OnChanged(object source, FileSystemEventArgs e)
  {
    // 指定当文件改变、创建或者删除的时候需要做的事情
    Console.WriteLine("File: {0} {1}!", e.FullPath, e.ChangeType);
  }

  static void OnRenamed(object source, RenamedEventArgs e)
  {
    // 指定当文件重命名的时候需要做的事情
    Console.WriteLine("File: {0} renamed to {1}", e.OldFullPath, e.FullPath);
  }
```

为了测试该程序，运行应用程序并且打开Windows资源管理器。试着重命名文件，新建一个*.txt文件，删除一个*.txt文件等，我们就会看到这个控制台应用程序输出了各种关于MyFolder文件夹内文本文件当前状态的信息：

```
***** The Amazing File Watcher App *****
Press 'q' to quit app.
File: C:\MyFolder\New Text Document.txt Created!
File: C:\MyFolder\New Text Document.txt renamed to
C:\MyFolder\Hello.txt
File: C:\MyFolder\Hello.txt Changed!
File: C:\MyFolder\Hello.txt Changed!
File: C:\MyFolder\Hello.txt Deleted!
```

源代码　MyDirectoryWatcher项目的源代码位于Chapter 20子目录下。

这就是本章对于.NET平台下基础I/O操作的介绍。你肯定会在很多应用程序中使用这些技术，而且你可能还会发现对象序列化服务可以显著地简化对大量数据的持久化。

20.13　对象序列化

术语序列化（serialization）描述了持久化（可能还包括传输）一个对象的状态到流（如文件流和内存流）的过程。被持久化的数据次序包括所有以后需要用来重建（即反序列化）对象状态所必需的

信息。使用这种技术，用最小花费来保存海量的（各种格式的）数据就变得轻而易举了。实际上，在很多情况下，使用序列化服务保存应用程序数据，相对直接使用System.IO命名空间的读取器/编写器，减少了很多的麻烦。

举例来说，设想你建立了一个基于GUI的桌面应用程序，并希望提供一种方法给最终用户保存他们偏好（窗口颜色和字号等）的界面。为此，你可能定义了一个名为UserPrefs的类来封装20个字段数据。如果使用System.IO.BinaryWriter类型，需要人工保存UserPrefs对象的每个字段成员。同样，当想从文件把这些数据导入内存的时候，需要使用System.IO.BinaryReader并且（再一次）人工地读入每个值以重新生成一个新的UserPrefs对象。

虽然这是可以实现的，但若为UserPrefs类标识[Serializable]特性将可以轻易地节省大量的时间。

```
[Serializable]
public class UserPrefs
{
  public string WindowColor;
  public int FontSize;
}
```

这时，对象全部的状态可以通过下面几行代码来持久化。暂时不用担心细节，考虑下面的Main()方法：

```
static void Main(string[] args)
{
  UserPrefs userData= new UserPrefs();
  userData.WindowColor = "Yellow";
  userData.FontSize = 50;

  // BinaryFormatter以二进制格式持久化状态数据
  // 需要导入System.Runtime.Serialization.Formatters.Binary以便访问BinaryFormatter
  BinaryFormatter binFormat = new BinaryFormatter();

  // 现在将对象保存到一个本地文件中
  using(Stream fStream = new FileStream("user.dat",
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, userData);
  }
  Console.ReadLine();
}
```

虽然使用.NET对象序列化保存对象非常简单，但幕后的调用过程却非常复杂。例如，当一个对象被持久化到流时，所有的相关数据（基类、包含的对象等）也会被自动序列化，因此，假如你想持久化一个派生类，那么继承链（即基类，基类的基类）上的所有数据都会被包括进来，后面你会看到，一组相关的对象使用对象图来表现。

.NET序列化服务也允许用多种格式来保存一个对象图。先前的示例代码使用了BinaryFormatter类型，所以UserPrefs对象的状态被保存为紧凑的二进制格式。你也可以使用其他类型将对象图保存为简单对象访问协议（SOAP）或XML格式。当希望确保你的持久化对象跨越操作系统、语言和结构进行传递时，这些格式是很有用的。

说明 WCF序列化对象的机制稍有不同。它使用[DataContract]和[DataMember]特性。你将在第25章中学习这些内容。

最后，要知道对象图可以持久化为任意的System.IO.Stream派生类型。在先前的示例中，通过FileStream类型将UserPrefs对象持久化到一个本地文件。但是，如果你想保存对象到内存中，可以使用MemoryStream类型。关键是数据的顺序要正确地代表图中对象的状态。

对象图的作用

前面提到过，当对象被序列化时，CLR将处理所有相关的对象，一组关联对象被总称为一个对象图。对象图提供一种很简明的方式来记载一组对象如何相互引用对方。注意，对象图没有指名面向对象关系中的“is-a”或“has-a”关系。相反，它们用箭头来表示“需要”和“依赖”的关系。

在对象图中的每个对象被赋予一个独有的数值类型的值。谨记：这个数可任意地赋给对象图中的成员，对外部世界没有真正意义。一旦所有的对象都赋予了数值，对象图就可以记录每个对象的整个依赖关系。

举一个简单的例子，比如建立一组类模拟一些汽车。有一个基类名称是Car，它配有（“has-a”）Radio，另外一个名为JamesBondCar的类扩展Car基类。图20-4显示了一个模拟这些关系的可能的对象图。

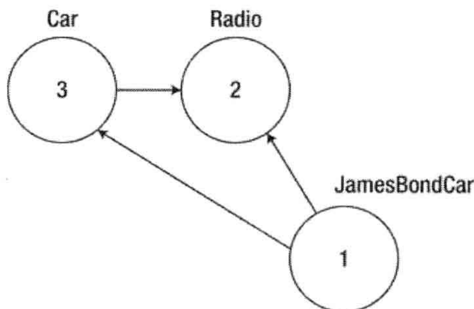


图20-4 简单的对象图

读取对象图时，可以使用短语依赖于（depend on）或引用（refer to）来表示连接箭头。因而，在图20-4里可以看到Car类引用了Radio类（有“has-a”关系）。JamesBondCar类引用了Car类（是“is-a”的关系），也引用了Radio类（因为它继承了这个受保护的成员变量）。

当然，CLR不会在内存中画图来表示相关对象。图20-4中记载的关系会用类似下面这样的、更具有数学含义的公式来描述：

```
[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]
```

如果解析这个公式，会再次看到对象3（Car类）依赖于对象2（Radio类），对象2（Radio类）像一个孤僻的人，不依赖任何人。最后，对象1（JamesBondCar类）依赖于对象3和对象2。在各种情况下，序列化或反序列化JamesBondCar类的实例时，对象图确认Radio类型和Car类型也参与了这个过程。

序列化过程的美妙之处在于表示各个对象之间相互关系的图形是在幕后自动建立的。本章后面将会看到，如果希望更多地介入到某个对象图的构建，通过使用特性和接口来自定义序列化过程也是可以做到的。

说明 严格说来，XmlSerializer类型（本章后面会介绍）不使用对象图来持久化状态，然而，这个类型会以可预知的方式序列化和反序列化相关对象。

20.14 为序列化配置对象

为了让一个对象支持.NET序列化服务，用户需要做的只是为每一个关联的类（或结构）加上[Serializable]特性，真的就是这样简单。如果你觉得给定的类型有一些成员数据不能（或可能不）参与到序列化配置中，可以在这些域前加上[NonSerialized]特性。如果在可被序列化的类中有成员变量不需要保存（比如，固定的值、随机值、瞬态数据等）并且希望减小持久化图的大小，这样做是很有用的。

20.14.1 定义可序列化的类型

创建一个新的控制台应用程序SimpleSerialize，其中的Radio类被标记为[Serializable]，除了一个成员变量（radioID）例外，它被标记为[NonSerialized]，因此radioID类将不会被持久化到指定的数据流中：

```
[Serializable]
public class Radio
{
  public bool hasTweeters;
  public bool hasSubWoofers;
  public double[] stationPresets;

  [NonSerialized]
  public string radioID = "XF-552RR6";
}
```

接下来，添加另外两个类类型来表示JamesBondCar和Car基类。JamesBondCar类和Car基类也标记为[Serializable]，并且定义了下列字段数据：

```
[Serializable]
public class Car
{
  public Radio theRadio = new Radio();
  public bool isHatchBack;
}

[Serializable]
public class JamesBondCar : Car
{
  public bool canFly;
  public bool canSubmerge;
}
```

注意，[Serializable]特性不能被继承。因此，如果从被标记为[Serializable]的类派生一个类，子类也必须被标记为[Serializable]，否则它不能被持久化。实际上，对象图中的所有对象必须标上[Serializable]特性。如果试图使用BinaryFormatter或者SoapFormatter序列化一个非序列化的对象，在运行时将会收到一个SerializationException（序列化异常）的提示。

20.14.2　公共字段、私有字段和公共属性

注意，为了简化范例，上面用到每一个类中的字段数据都被定义为公共的。当然，从面向对象的观点看，用公共属性公开私有字段数据更受欢迎。同样，为了简化的需要，上面的示例也没有定义构造函数，因此所有未被赋值的字段数据将接收期望的默认值。

除了面向对象的设计原则之外，读者可能想知道不同的格式化方法指望类型的字段数据要如何定义，才能被序列化为流。答案是，这取决于具体情况。如果使用BinaryFormatter或SoapFormatter持久化一个对象，完全没有区别。这些类型被编程为序列化一个类型的所有可序列化的字段，不管它是公共字段、私有字段还是通过公共属性公开的私有字段。回想一下，如果有一些不想被持久化到对象图中的数据点，可以有选择地把公共或私有字段标记为[NonSerialized]，像对Radio类型中的字符串域所做的那样。

如果使用XmlSerializer类型，情况就大不同。这些类型只有字段数据的公共块或拥有公共属性的私有数据可以被序列化。不是通过属性公开的私有数据将被忽略，例如下面可序列化的Person类型：

```
[Serializable]
public class Person
{
  // 公共字段
  public bool isAlive = true;

  // 私有字段
  private int personAge = 21;

  // 公共属性/私有数据
  private string fName = string.Empty;
  public string FirstName
  {
    get { return fName; }
    set { fName = value; }
  }
}
```

如果是由BinaryFormatter或SoapFormatter进行处理，我们就会发现isAlive、personAge以及fName都保存到了所选的流中。然而，XmlSerializer不会保存personAge的值，因为这段私有数据没有封装为类型属性。如果你希望使用XmlSerializer来持久化用户年龄，就需要把字段定义为公共的或使用公共属性来封装私有字段。

20.15　选择序列化格式化程序

一旦将类型配置为参与.NET序列化，接下来就是选择当持久化对象图时使用哪种格式（二进制、SOAP或XML），有以下3种选择：

- BinaryFormatter；
- SoapFormatter；
- XmlSerializer。

BinaryFormatter类型使用紧凑的二进制格式将对象图序列化为一个流，这个类型在System.Runtime.Serialization.Formatters.Binary命名空间中定义，后者是mscorlib.dll的一部分。因此，为了获得对这个类型的访问，需要指定下面的C# using指令：

```
// 获取对mscorlib.dll中的BinaryFormatter的访问
using System.Runtime.Serialization.Formatters.Binary;
```


SoapFormatter类型将对象图表示为一个SOAP消息（传递消息到Web服务或从Web服务传递消息的标准XML格式）。该类型定义在System.Runtime Serialization.Formatters.Soap命名空间中，该命名空间被定义在一个程序集内。因此，要格式化对象图为一个SOAP消息，必须使用Visual Studio的Add Reference对话框设定引用指向System.Runtime.Serialization.Formatters.Soap.dll并指定下列的C# using指令：

```
// 必须引用System.Runtime.Serialization.Formatters.Soap.dll
using System.Runtime.Serialization.Formatters.Soap;
```

最后，如果希望将对象图持久化为一个XML文档，需要用到XmlSerializer类型。要使用这个类型，需要指定使用System.Xml.Serialization命名空间，并设置对程序集System.Xml.dll的引用。幸好所有的Visual Studio项目模板自动引用了System.Xml.dll，因此只需要使用下列命名空间：

```
// 定义在System.Xml.dll
using System.Xml.Serialization;
```

20.15.1 IFormatter和IRemotingFormatting接口

不管选择哪种格式化程序来序列化对象，都要知道它们直接派生于System.Object，因此并不从一个以序列化为中心的基类共享一组公共的成员。但是，BinaryFormatter和SoapFormatter类型通过实现IFormatter和IRemotingFormattter接口（XmlSerializer两者都不实现）都支持公共的成员。

System.Runtime.Serialization.IFormatter定义了核心的Serialize()和Deserialize()方法，Serialize()和Deserialize()方法将做复杂的工作完成对象图和指定流之间的转换。除了这些成员，IFormatter还定义了一些在后台使用的实现类型的属性：

```
public interface IFormatter
{
  SerializationBinder Binder { get; set; }
  StreamingContext Context { get; set; }
  ISurrogateSelector SurrogateSelector { get; set; }
  object Deserialize(Stream serializationStream);
  void Serialize(Stream serializationStream, object graph);
}
```

System.Runtime.Remoting.Messaging.IRemotingFormatter接口（被.NET远程处理层内部控制）重载了Serialize()和Deserialize()成员使风格更适合于分布式持久化。注意IRemotingFormatter派生于更基本的IFormatter接口：

```
public interface IRemotingFormatter  : IFormatter
{
  object Deserialize(Stream serializationStream, HeaderHandler handler);
  void Serialize(Stream serializationStream, object graph, Header[] headers);
}
```

尽管在大多数序列化工作时不需要直接与这些接口发生交互，回想一下，基于接口的多态性允许使用一个IFormatter引用来保持一个BinaryFormatter或SoapFormatter的实例。因此，如果希望建立一个方法使用这两个类中的一个来序列化一个对象图，可以编写下列语句：

```
static void SerializeObjectGraph(IFormatter itfFormat,
                                 Stream destStream, object graph)
{
  itfFormat.Serialize(destStream, graph);
}
```

20.15.2　在格式化程序中的类型保真

在3种格式化程序中，最明显的不同是对象图被持久化为不同的流（二进制、SOAP或XML）的方式。读者一定要意识到其中更细微的差别，特别是格式化程序如何应对类型保真（type fidelity）。当使用BinaryFormatter类型时，不仅仅是将对象图中对象的字段数据进行持久化，而且也持久化每个类型的完全限定名称和定义程序集的完整名称（名称、版本、公钥标记和区域性）。这些数据使BinaryFormatter在希望用值（例如以一个完整的副本）跨越.NET应用程序机器边界传递对象时成为理想的选择。

SoapFormatter通过使用XML命名空间来持久化原始程序集的跟踪。例如，本章之前的Person类型。如果类型使用SOAP消息进行持久化，我们就会发现Person的开始元素使用生成的xmlns进行限定。考虑如下片段的定义，特别注意a1 XML命名空间：

```
<a1:Person id="ref-1" xmlns:a1=
  "http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/MyApp%2C%20
  Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
  <isAlive>true</isAlive>
  <personAge>21</personAge>
  <fName id="ref-3">Mel</fName>
</a1:Person>
```

另一方面，XmlSerializer不会试图保存完全的类型保真，因此也不记录类型完全限定名称或起源的程序集。因此可能乍一看好像有些限制，但XML序列化用于标准的.NET Web服务，可被任何平台（不仅仅是.NET）中的客户端调用。这意味着没有必要序列化完整的.NET类型元数据。下面是Person类型可能的XML表示方法：

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <isAlive>true</isAlive>
  <PersonAge>21</PersonAge>
  <FirstName>Frank</FirstName>
</Person>
```

如果希望持久化的对象图可以被任意操作系统（Windows、Mac OS X和各种Linux类系统）、应用程序框架（.NET、Java EE、COM等）或编程语言使用，就不要保持完整的类型保真，因为不能假设所有可能的接收方都能理解.NET专有的数据类型。基于此，当希望尽可能延伸持久化对象图的使用范围时，SoapFormatter和XmlSerializer是理想选择。

20.16 使用 BinaryFormatter 序列化对象

为说明持久化一个JamesBondCar的实例到一个物理文件中是多么简单，我们使用BinaryFormatter二进制类型。再次注意BinaryFormatter类型的两个关键方法：Serialize()和Deserialize()。

- Serialize()：将一个对象图按字节的顺序持久化到一个指定的流。
- Deserialize()：将一个持久化的字节顺序转化为一个对象图。

假设已经建立了JamesBondCar的一个实例，修改了一些状态数据，并想将间谍汽车[1]持久化到一个*.dat文件中。第一个任务是建立*.dat文件本身。这可以通过建立一个System.IO.FileStream类型的实例来完成。此刻，仅仅建立一个BinaryFormatter的实例并传进FileStream和对象图进行持久化。考虑下面的Main()方法：

```
// 确保使用了SystemRuntime.Serialization.Formatters.Binary
// 和System.IO命名空间
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Serialization *****\n");

  // 建立一个JamesBondCar并设定状态
  JamesBondCar jbc = new JamesBondCar();
  jbc.canFly = true;
  jbc.canSubmerge = false;
  jbc.theRadio.stationPresets = new double[]{89.3, 105.1, 97.1};
  jbc.theRadio.hasTweeters = true;

  // 将car以二进制格式保存到指定文件中
  SaveAsBinaryFormat(jbc, "CarData.dat");
  Console.ReadLine();
}
```

SaveAsBinaryFormat()方法按如下所示实现：

```
static void SaveAsBinaryFormat(object objGraph, string fileName)
{
  // 将对象以二进制保存到一个名为CarData.dat的文件
  BinaryFormatter binFormat = new BinaryFormatter();

  using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in binary format!");
}
```

可见，BinaryFormatter.Serialize()方法是一个负责生成对象图并将字节顺序移动到流的派生类型的成员。在这个例子中，流碰巧是一个物理文件。然而，也可以序列化对象类型为任意流的派生类型，例如放置到内存、网络流中。

在运行程序之后，我们就可以转到当前项目\bin\Debug文件夹查看表示JamesBondCar实例的CarData.dat文件的内容了。图20-5显示了在Visual Studio中打开的这个文件。

---

① 前面的JamesBondCar是“007中邦德的车”的意思。——译者注

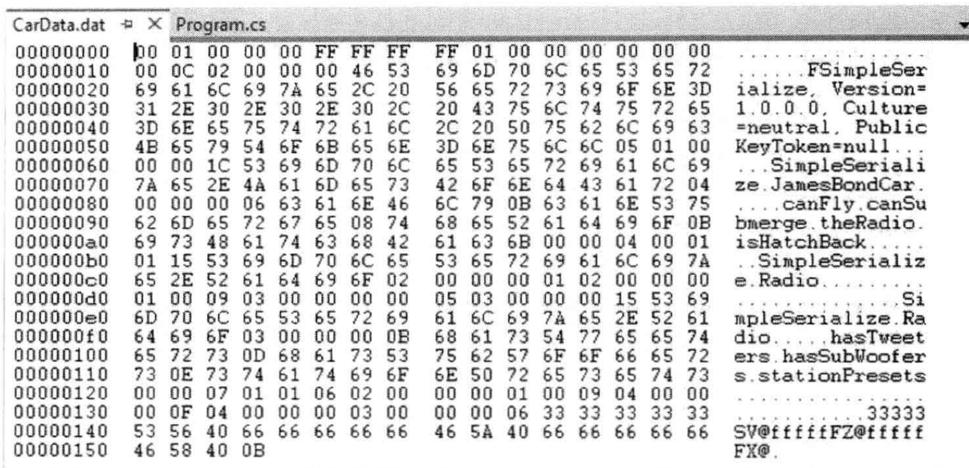


图20-5　使用BinaryFormatter对JamesBondCar进行序列化

使用BinaryFormatter反序列化对象

现在假定你在考虑从二进制文件中读取被持久化的JamesBondCar并将其恢复到一个对象变量中。一旦以编程方式打开CarData.dat（通过File.OpenRead()方法）文件，只需要调用BinaryFormatter的Deserialize()方法。要知道，Deserialize()返回一个基本的System.Object类型，所以需要强制施加外部转换，如下面显示的：

```
static void LoadFromBinaryFile(string fileName)
{
  BinaryFormatter binFormat = new BinaryFormatter();

  // 从二进制文件中读取JamesBondCar对象
  using(Stream fStream = File.OpenRead(fileName))
  {
    JamesBondCar carFromDisk =
      (JamesBondCar)binFormat.Deserialize(fStream);
    Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
  }
}
```

注意，如果我们调用Deserialize()，需要传入表示持久的对象图位置的Stream的派生类型。在把对象转换回正确类型之后，我们就可以发现状态数据是我们保存对象时的那个状态点。

20.17　使用 SoapFormatter 序列化对象

下一个格式化程序的选择是SoapFormatter类型。SoapFormatter类型将把对象图持久化为一个SOAP消息。简而言之，SOAP定义了一个标准的过程，在这个过程中可以用与平台和操作系统无关的方式调用方法。

假定你引用了System.Runtime.Serialization.Formatters.Soap.dll程序集，并使用了System.Runtime.

Serialization.Formatters.Soap命名空间，只需将出现的每个BinaryFormatter替换为SoapFormatter就可以持久化并接收JamesBondCar为一个SOAP消息。参考下面Program类的新方法，它将一个对象序列化为一个SOAP格式的本地文件：

```
// 确保使用了System.Runtime.Serialization.Formatters.Soap并引用了
// System.Runtime.Serialization.Formatters.Soap.dll
static void SaveAsSoapFormat (object objGraph, string fileName)
{
  // 将对象以SOAP格式保存到CarData.soap文件中
  SoapFormatter soapFormat = new SoapFormatter();

  using(Stream fStream = new FileStream(fileName,
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    soapFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in SOAP format!");
}
```

和之前一样，仅仅使用Serialize()和Deserialize()方法将对象图移入和移出流。如果从Main()调用这方法，并运行程序，将打开一个产生结果的*.soap文件。可以定位到标记了当前JamesBondCar的状态值的XML元素上，也可以通过#ref标记定位到对象图中对象间的关系上，如图20-6所示。

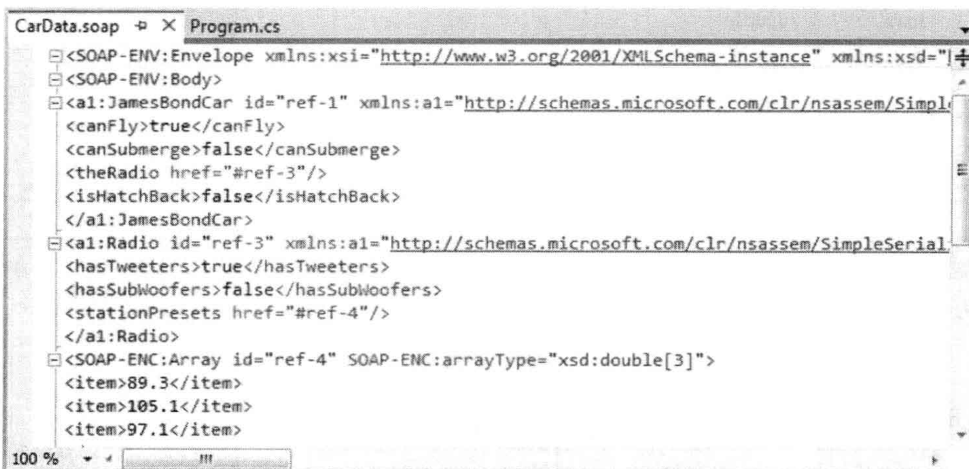


图20-6 使用SoapFormatter对JamesBondCar进行序列化

20.18 使用 XmlSerializer 序列化对象

除了SOAP和二进制格式化程序外，System.Xml.dll程序集提供了第三种格式化程序：System.Xml.Serialization.XmlSerializer。与XML数据被包含在一个SOAP消息中相反，该方式可以被用来将给定对象的公共状态持久化为一个纯XML。使用这种类型与使用SoapFormatter或BinaryFormatter类型有一点不同。参考下面的代码，假定使用了System.Xml.Serialization命名空间：

```
static void SaveAsXmlFormat(object objGraph, string fileName)
{
  // 将对象以XML格式保存到CarData.xml文件中
  XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar));

  using(Stream fStream = new FileStream(fileName,
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    xmlFormat.Serialize(fStream, objGraph);
  }
  Console.WriteLine("=> Saved car in XML format!");
}
```

关键的不同点是XmlSerializer类型需要你指定类型信息表示要序列化的类。如果查看新生成的XML文件（假设你调用了Main()中的新方法），可以看到如下所示的XML数据：

```
<?xml version="1.0"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <theRadio>
    <hasTweeters>true</hasTweeters>
    <hasSubWoofers>false</hasSubWoofers>
    <stationPresets>
      <double>89.3</double>
      <double>105.1</double>
      <double>97.1</double>
    </stationPresets>
    <radioID>XF-552RR6</radioID>
  </theRadio>
  <isHatchBack>false</isHatchBack>
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

说明　XmlSerializer要求对象图中的所有序列化类型支持默认的构造函数（所以如果定义自定义的构造函数，一定记得要把它加回来）。如果没有，将在运行时收到InvalidOperationException。

控制生成的XML数据

如果读者有使用XML技术的背景，就知道确认XML文档中的元素符合一套建立数据有效性的规则往往是很关键的。要理解有效的XML文档并不意味着XML元素都符合句法（比如，所有打开的元素都必须有结束元素）。准确地说，有效的文档符合事先达成共识的格式化规则（例如，X字段必须被表示成一个特性而不是一个子元素），这个标准通常由XML架构或文档类型定义（DTD）文件定义。

默认情况下，XmlSerializer将所有公有字段/属性序列化为XML元素而不是XML特性。如果希望控制XmlSerializer如何生成XML文档，可以用取自System.Xml.Serialization命名空间的任意数量的附加特性来修饰你的类型。表20-12记录了部分（而不是全部）.NET特性，这些特性影响XML数据被编码为一个流的方式。

表20-12 System.Xml.Serialization命名空间中的部分特性

特　性	作　用
[XmlAttribute]	可以在类的公共字段上使用这个.NET特性，它告诉XmlSerializer将数据作为XML特性（不是子元素）进行序列化
[XmlElement]	字段或属性将作为XML元素被序列化
[XmlEnum]	枚举成员的元素名称
[XmlRoot]	该特性控制根元素如何被构造（命名空间和元素名称）
[XmlText]	属性或字段将被序列化为XML文本（即根元素中开始标签和结束标签之间的内容）
[XmlType]	XML类型的名称和命名空间

下面是一个简单的例子，首先考虑JamesBondCar的字段数据当前如何被序列化为XML：

```
<?xml version="1.0" encoding="utf-8"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

如果希望指定一个自定义的XML命名空间限定修饰JamesBondCar，还要将canFly和canSubmerge值编码为XML特性，也可以依照下面这样来修改JamesBondCar的C#定义：

```
 [Serializable, XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
  [XmlAttribute]
  public bool canFly;
  [XmlAttribute]
  public bool canSubmerge;
}
```

这样将生成下面的XML文档（注意打开的<JamesBondCar>元素）：

```
<?xml version="1.0"""?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  canFly="true" canSubmerge="false"
  xmlns="http://www.MyCompany.com">
...
</JamesBondCar>
```

当然，有大量其他的特性可被用来控制XmlSerializer生成最终的XML文档的方式。如果你希望了解所有的选项，请在.NET Framework 4.5 SDK文档中查找System.Xml.Serialization命名空间。

20.19 序列化对象集合

现在已经看到了如何将一个对象持久化为一个流，下面检验如何保存一组对象。读者可能注意到了，IFormatter接口中的Serialize()方法不提供指定任意数量对象的方法（只能是一个System.Object）。相关的是，Deserialize()方法的返回值同样也是一个System.Object（同样的限制也适用于XmlSerializer）：

```
public interface IFormatter
{
...
  object Deserialize(Stream serializationStream);
  void Serialize(Stream serializationStream, object graph);
}
```

回想一下，System.Object实际上表现了一个完整的对象图。基于此，如果你传递进来一个被标记为[Serializable]的对象并且还包含了其他[Serializable]对象，整个对象集可以立刻被持久化。幸运的是，大多数在System.Collections和System.Collections.Generic命名空间内的类型已经被标记为[Serializable]。因此，如果你希望对一组对象进行持久化，只需要添加这组对象到容器（比如普通数组、ArrayList或List<T>）中并序列化对象为你选择的流就可以了。

假设已经用一个双参数构造函数更新了JamesBondCar类，因此可以设置一些状态数据（注意已按照XmlSerializer的要求把默认的构造函数加了回去）：

```
[Serializable,
 XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
  public JamesBondCar(bool skyWorthy, bool seaWorthy)
  {
    canFly = skyWorthy;
    canSubmerge = seaWorthy;
  }
  // XmlSerializer需要一个默认的构造函数
  public JamesBondCar(){}
...
}
```

有了这些，就可以按照以下方式持久化任何数目的JamesBondCars：

```
static void SaveListOfCars()
{
  // 现在持久化一个JamesBondCar的List<T>
  List<JamesBondCar> myCars = new List<JamesBondCar>();
  myCars.Add(new JamesBondCar(true, true));
  myCars.Add(new JamesBondCar(true, false));
  myCars.Add(new JamesBondCar(false, true));
  myCars.Add(new JamesBondCar(false, false));

  using(Stream fStream = new FileStream("CarCollection.xml",
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    XmlSerializer xmlFormat = new XmlSerializer(typeof(List<JamesBondCar>));
    xmlFormat.Serialize(fStream, myCars);
  }
  Console.WriteLine("=> Saved list of cars!");
}
```

重复一下，因为使用了XmlSerializer，所以需要为每个子对象在根对象（本例是List<JamesBondCar>）中指定类型信息。如果使用了BinaryFormatter或SoapFormatter类型，逻辑可能更简单，举例来说：

```
static void  SaveListOfCarsAsBinary()
{
  // 将ArrayList对象(myCars)保存为二进制
  List<JamesBondCar> myCars = new List<JamesBondCar>();
```

```
  BinaryFormatter binFormat = new BinaryFormatter();
  using(Stream fStream = new FileStream("AllMyCars.dat",
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    binFormat.Serialize(fStream, myCars);
  }
  Console.WriteLine("=> Saved list of cars in binary!");
}
```


源代码　SimpleSerialize应用程序的源代码位于Chapter 20子目录下。

20.20　自定义 Soap/Binary 序列化过程

大多数情况下，由.NET平台提供的默认序列化方案都可以满足需要，只需要应用[Serializable]特性并将对象图传送给选择的格式化程序即可。但是在一些情况下，你也许希望在序列化过程中更多地干预构造和处理目录树的过程。比如，可能你有一个商业规则，规定所有的字段数据都必须被持久化为大写格式，或者可能你希望添加额外的一些数据位到流中，而该流不映射到正在被持久化的对象的字段上（时间戳、唯一标识符或其他）。

当希望更多地参与对象序列化过程时，System.Runtime.Serialization命名空间提供了以下几个类型。表20-13列出了核心类型。

表20-13　System.Runtime.Serialization命名空间核心类型

类　型	作　用
ISerializable	在[Serializable]类型上实现这个接口来控制序列化和反序列化
ObjectIDGenerator	该类型为对象图中的成员生成唯一标识符
[OnDeserialized]	允许指定的方法在对象被反序列化后立即被调用
[OnDeserializing]	允许指定的方法在对象被反序列化之前被调用
[OnSerialized]	允许指定的方法在对象被序列化后立即被调用
[OnSerializing]	允许指定的方法在对象被序列化之前被调用
[OptionalField]	允许在类型中定义一个可能在指定流中丢失的字段
[SerializationInfo]	本质上，这个类是一个“属性包”，持有在序列化过程中表示对象状态的名称/值对

20.20.1　深入了解对象序列化

在检验可以用来自定义序列化过程的不同方法之前，有必要深入了解在后台发生了什么。当BinaryFormatter序列化一个对象图时，它负责传送下面的信息到指定的流。

- 在对象图中对象的完全限定名（如MyApp.JamesBondCar）。
- 定义对象图的程序集名称（如MyApp.exe）。
- SerializationInfo类的一个实例，包含了所有由对象图成员保存的所有描述性数据。

在反序列化过程中，BinaryFormatter使用相同的信息建立对象的一模一样的副本，使用从基层流中提取的信息，SoapFormatter使用的过程也和序列化相似。

说明　回想一下：XmlSerializer不能持久化一个类型的完全限定名称或定义的程序集的名称以保证对象状态尽可能轻巧。这些类型仅仅与持久化公开的字段数据相关。

除了因为要将需要的数据移入或移出流，格式化程序也采用下面的基本办法来分析对象图中的成员。

- 做一个检测来确定对象是否被标记了[Serializable]特性。如果对象没有标记，则抛出SerializationException。
- 如果对象被标记了[Serializable]，做一个检测来确定对象是否实现了ISerializable接口，如果是，GetObjectData()方法会在对象上调用。
- 如果对象没有实现ISerializable，使用默认的序列化过程，序列化所有没有被标记为[NonSerialized]的字段。

除了确定类型是否支持ISerializable外，格式化程序也负责发现类型是否支持被[OnSerializing]、[OnSerialized]、[OnDeserializing]或[OnDeserialized]等特性修饰的成员。我们稍后将学习这些特性的作用，首先看看ISerializable的作用。

20.20.2　使用ISerializable自定义序列化

被标记了[Serializable]的对象拥有了实现ISerializable接口的选项。这样可以更为深入地“涉足”序列化过程并可以执行任何前数据和后数据格式化。

说明　从.NET 2.0发布之后，自定义序列化的过程推荐使用序列化特性（后面会介绍）。然而，如果要维护既有系统的话，了解ISerializable也很重要。

这个接口十分简单，因为它只定义了一个简单的方法GetObjectData()：

```
// 当希望逆转序列化过程时，就实现ISerializable
public interface ISerializable
{
  void GetObjectData(SerializationInfo info,
    StreamingContext context);
}
```

GetObjectData()方法在序列化过程期间被给定的格式化程序自动调用。该方法的实现用一系列名称/值对填充了输入的SerializationInfo参数，这些名称/值对（通常）映射到将被持久化对象的字段数据上。除了一小组允许类型获取和设置类型名称的属性、定义的程序集和成员计数外，SerializationInfo还定义了多种不同的重载的AddValue()方法。这里是部分代码：

```
public sealed class SerializationInfo
{
  public SerializationInfo(Type type, IFormatterConverter converter);
  public string AssemblyName { get; set; }
```

```
  public string FullTypeName { get; set; }
  public int MemberCount { get; }
  public void AddValue(string name, short value);
  public void AddValue(string name, ushort value);
  public void AddValue(string name, int value);
...
}
```


实现ISerializable接口的类型也必须定义一个带有下面签名的特殊构造函数：

```
// 你必须提供一个自定义的、带有这个签名的构造函数，以允许运行库引擎设置对象的状态
[Serializable]
class SomeClass : ISerializable
{
  protected SomeClass (SerializationInfo si, StreamingContext ctx) {...}
  ...
}
```

注意构造函数的可见性被设置为受保护的，这是允许的，因为格式化程序可以访问这个成员而不管它是否可见。这个特殊的构造函数特意被标记为受保护的（私有），以确保不熟悉对象技术的新用户不能用这种方式建立对象。如你所见，构造函数的第一个参数是一个SerializationInfo类型（参考前文）的实例。

这个特殊的构造函数的第二个参数是StreamingContext类型，它包含了序列化流的源的有关信息。这个类型信息量最大的成员是State属性，它表示StreamingContextStates枚举中的值。该枚举的值表示当前流的基本构成。

事实上，除非去实现一些低级的自定义远程服务，否则很少需要直接处理这个枚举值。下面是StreamingContextStates枚举可能的名字（细节请参考.NET Framework 4.5 SDK文档）：

```
public enum StreamingContextStates
{
  CrossProcess,
  CrossMachine,
  File,
  Persistence,
  Remoting,
  Other,
  Clone,
  CrossAppDomain,
  All
}
```

为说明使用ISerializable自定义序列化的过程，假设创建了新控制台应用程序CustomSerialization，它定义了一个类类型，其中包含两个字符串类型的数据。此外，假定你必须确定要被序列化为流的字符串对象全是大写字母并且从流中反序列化为小写字母。为满足这个规则，可以像下面这样实现ISerializable（一定要导入System.Runtime.Serialization命名空间）：

```
[Serializable]
class StringData : ISerializable
{
  private string dataItemOne = "First data block";
  private string dataItemTwo= "More data";

  public StringData(){}
  protected StringData(SerializationInfo si, StreamingContext ctx)
  {
```

```
    // 从流中得到合并的成员变量
    dataItemOne = si.GetString("First_Item").ToLower();
    dataItemTwo = si.GetString("dataItemTwo").ToLower();
  }

  void ISerializable.GetObjectData(SerializationInfo info, StreamingContext ctx)
  {
    // 用格式化数据填充SerializationInfo对象
    info.AddValue("First_Item", dataItemOne.ToUpper());
    info.AddValue("dataItemTwo", dataItemTwo.ToUpper());
  }
}
```

注意，当在GetObjectData()方法里填充SerializationInfo类型时，不需要为数据点取与类型内部成员变量一样的名字。如果需要从持久化格式中更进一步地解耦类型的数据，显而易见这是很有用的。可是，要知道你需要使用和分配在GetObjectData()中的相同的名称从私有构造函数中得到值。

为测试自定义序列化，假定已经使用SoapFormatter持久化了一个MyStringData的实例（因此更新程序集引用并相应地导入命名空间）：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Serialization *****");

  // 这个类型实现了ISerializable
  StringData myData = new StringData();

  // 以SOAP格式保存到本地文件中
  SoapFormatter soapFormat = new SoapFormatter();
  using(Stream fStream = new FileStream("MyData.soap",
    FileMode.Create, FileAccess.Write, FileShare.None))
  {
    soapFormat.Serialize(fStream, myData);
  }
  Console.ReadLine();
}
```

当查看得到的*.soap文件时，注意到字符串字段已经真正地被持久化为大写字母了。

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>

  <a1:StringData id="ref-1" ...>
    <First_Item id="ref-3">FIRST DATA BLOCK</First_Item>
    <dataItemTwo id="ref-4">MORE DATA</dataItemTwo>
  </a1:StringData>
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

20.20.3 使用特性定制序列化

尽管实现ISerializable接口来定制序列化过程还是可能的，但是自.NET 2.0发布以来，定制序列

化过程的首选方式是定义一些具有新的序列化相关特性的方法（这些特性如[OnSerializing]、[OnSerialized]、[OnDeserializing]或[OnDeserialized]）。使用这些特性，减少了实现ISerializable的麻烦，因为不需要手动与输入的SerializationInfo参数交互，而是在格式化程序操作该类型时直接修改状态数据。

说明 这些序列化特性定义在System.Runtime.Serialization命名空间中。

当应用这些特性时，必须定义方法接收一个StreamingContext参数并返回空（否则，将收到一个运行时异常）。注意，不需要说明每一个与序列化有关的特性，并且可以仅仅只处理你感兴趣而截取的序列化阶段。举例来说，这里有一个新的[Serializable]类型，与StringData有相同的需求，这次使用[OnSerializing]和[OnDeserialized]特性来解决：

```
[Serializable]
class MoreData
{
  private string dataItemOne = "First data block";
  private string dataItemTwo= "More data";

  [OnSerializing]
  private void OnSerializing(StreamingContext context)
  {
    // 在序列化过程中就得到调用
    dataItemOne = dataItemOne.ToUpper();
    dataItemTwo = dataItemTwo.ToUpper();
  }

  [OnDeserialized]
  private void OnDeserialized(StreamingContext context)
  {
    // 一旦反序列化过程结束，就得到调用
    dataItemOne = dataItemOne.ToLower();
    dataItemTwo = dataItemTwo.ToLower();
  }
}
```

如果序列化这个新类型，将再次发现数据被持久化为大写字母并被反序列化为小写字母。

源代码 CustomSerialization项目的源代码位于Chapter 20子目录下。

这些示例给我们介绍了有关对象序列化服务，包括以各种方式定制过程的核心细节。我们已经看到，序列化和反序列化过程使得持久化大量数据变得很简单，并且结合System.IO命名空间中的各种读取/写入类一起使用也更简单。

20.21 小结

本章一开始介绍了如何使用Directory(Info)和File(Info)类型。通过学习，我们知道了怎么通过这些类型操作硬盘上的物理文件或目录。接着我们研究了许多从Stream抽象类派生的类型。因为从

Stream派生的类型操作的是原始字节流，所以system.IO命名空间提供了很多读取器/编写器类型（如StreamWriter、StringWriter和BinaryWriter）来简化这个读写过程。同样，我们也了解了DriveType提供的功能，并且学习了如何使用FileSystemWatcher类型来监控文件以及如何使用异步方式和流进行交互。

本章还介绍了对象序列化服务的主题。可见，.NET平台使用一个对象图恰当地说明了要持久化到流中的完整的相关对象组。只要对象图中的每个成员标记了[Serializable]特性，数据就可以通过你选择的格式（二进制或SOAP）来进行持久化。

本章还介绍了使用两种可能的方法定制序列化过程。首先介绍了如何实现ISerializable接口（并支持一个特殊的私有构造函数），从而更深入地干预如何让格式化程序持久化所提供的数据。接下来，介绍了一组.NET新特性，它们简化了自定义序列化的过程。只需在带StreamingContext参数的成员上应用[OnSerializing]、[OnSerialized]、[OnDeserializing]或[OnDeserialized]特性，格式化程序就会相应地调用它们。

第21章

ADO.NET之一：连接层

为支持同数据库系统进行交互，.NET平台定义了许多命名空间，统称为ADO.NET。本章首先会总体介绍ADO.NET的作用，然后会重点讨论ADO.NET的数据提供程序。.NET平台支持许多数据提供程序，每一个都为与特定数据库管理系统（微软SQL Server、Oracle、MySQL等）进行通信做了优化。

理解了各个数据提供程序的常用功能后，我们会介绍数据提供程序工厂模式。使用System.Data.Common命名空间下的一些类型（和相关的App.config文件）后，只需要构建单个代码库就能实现动态选择和设置基本的数据提供程序，整个应用程序代码无需重新编译和部署。

可能最重要的是，本章会构建一个自定义的数据访问库程序集（AutoLotDAL.dll），它会封装在自定义数据库AutoLot上进行的各种操作。这个库会在第23章和第24章中进行扩展，并且会在本书后面几章中广泛使用。最后我们会研究数据库事务。

21.1 ADO.NET 的宏观定义

如果你曾使用过微软先前的一些基于COM的数据访问模型（Active Data Object，ADO）的话，你会发现ADO.NET已经和ADO没有什么关系了，而且ADO.NET已经超越了A、D、O这3个字母的概念。虽然不可否认，这两个体系之间还是有些联系的（比如说连接对象和命令对象的概念），但一些ADO中常见的类型（比如Recordset）在ADO.NET中已经没有了。此外，ADO.NET还新增了许多在传统ADO中找不到直接对应的新类型（比如数据适配器）。

传统ADO主要针对紧密连接的客户端/服务器端系统，而ADO.NET考虑到了断开连接式应用并且引进了DataSet。它代表任意数量的关联表，其中每个表都包含了行和列的集合的本地副本。使用DataSet的话，在断开数据库连接的情况下调用程序集（如Web页面或者桌面可执行程序）处理和更新它的内容，然后使用关联的*数据适配器*把修改后的数据提交回数据库。

从编程的角度来看，ADO.NET大部分由System.Data.dll核心程序集来表示。在这个二进制程序集中，我们可以找到许多命名空间（如图21-1所示），其中很多都表示某个ADO.NET数据提供程序（稍后会介绍）。

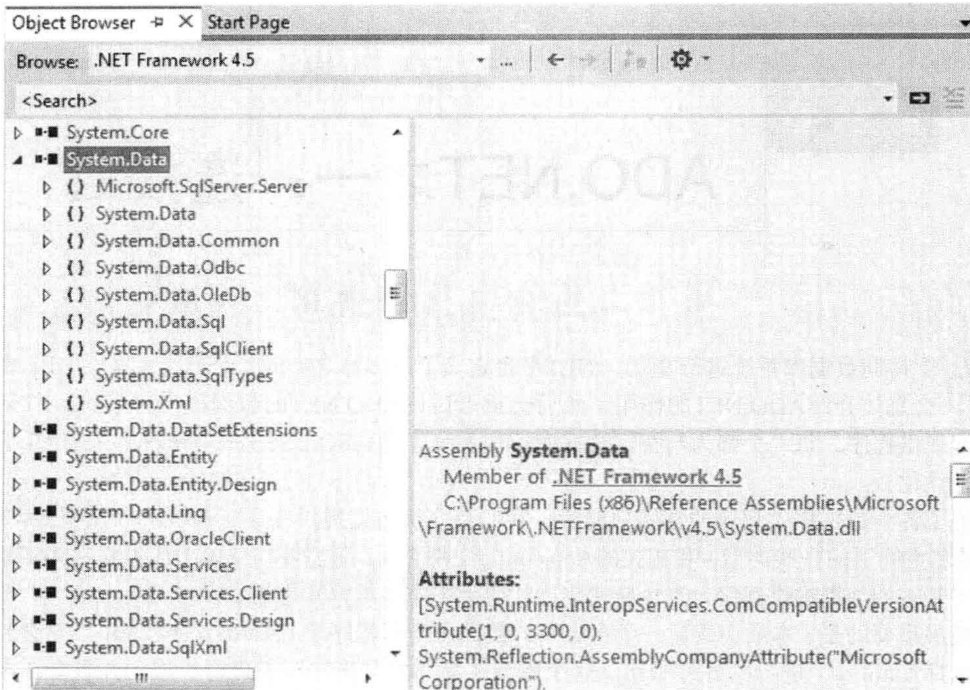


图21-1　System.Data.dll是核心ADO.NET程序集

其实，大多数Visual Studio项目模板会自动引用这个核心数据访问库。要知道，还有一些ADO.NET相关的程序集不在System.Data.dll中，我们需要通过Add Reference对话框来手动引入到当前项目中。

ADO.NET的三面

从概念上来说，ADO.NET类库有三种完全不同的方式来实现数据访问：连接式、断开式和通过Entity框架。当使用*连接式*的时候（本章的主题），你的代码需要显式连接或者断开基层数据源。用这种方式使用ADO.NET时，通常会用到连接对象、命令对象和数据读取器对象来实现这样的数据交互。

另一方面，断开式数据访问（详细介绍见第22章）允许通过一组`DataTable`对象（保存在`DataSet`中）来获取外部数据的一个客户端副本。当你通过相关的数据适配器对象来获取`DataSet`的时候，数据连接会自动打开或关闭。你可能也猜到了，这样能快速释放连接以便其他调用者使用，也极大增加了系统的可伸缩性。

一旦获取了一个`DataSet`后，就能在不需要花费网络流量的情况下随意修改内容。同样，如果你想把修改后的结果重新提交回数据库，需要再次使用数据适配器对象（关联一组SQL语句）来更新数据源，此时连接会为数据库更新重新打开并在结束操作后会立即被关闭。

最终在第23章，我们将介绍一个数据访问API，叫做Entity Framework（简称EF）。借助EF可以用封装了大量数据库底层细节的客户端对象与关系型数据库交互。同样，EF编程模型还可以使用LINQ to Entity语法，用强类型的LINQ查询与关系型数据库交互。

21.2 ADO.NET 数据提供程序

ADO.NET没有提供单一对象集来和多个数据库管理系统（DBMS）进行通信，而是提供了多种数据提供程序，每种为某个DBMS进行优化。这种方法的好处是，一来能以编程方式利用DBMS独有的一些特性，二来能直接和基层的DBMS引擎进行连接而不需要为不同的DBMS做中间的映射层。

简单来说，数据提供程序是一组定义在用于和特定的数据源类型进行通信的命名空间内的一组类型。不管你用的是哪种数据提供程序，它们都有一系列类来提供核心功能，表21-1列举了一些核心公共对象、它们的基类（都定义在System.Data.Common命名空间内）和它们实现的以数据为中心的接口（都定义在System.Data命名空间内）。

表21-1 ADO.NET数据提供程序的一些核心对象

对　象	基　类	实现的接口	作　用
Connection	DbConnection	IDbConnection	连接和断开数据源，提供了相关事务对象的访问
Command	DbCommand	IDbCommand	代表SQL查询语句或者存储过程名，同样提供了相关数据读取器对象的访问
DataReader	DbDataReader	IDataReader和IDataRecord	提供只读只向前形式的数据访问
DataAdapter	DbDataAdapter	IDataAdapter和IDbDataAdapter	在数据库和调用者之间传递DataSet，内置4个命令对象来实现数据的查询、插入、修改和删除操作
Parameter	DbParameter	IDataParameter和IDbDataParameter	在参数化查询中表示参数
Transaction	DbTransaction	IDbTransaction	实现数据库事务

尽管这些类的命名对于不同的数据提供程序不尽相同（比如SqlConnection和OracleConnection、OdbcConnection和MySqlConnection），但是它们都从相同的基类（就连接对象而言，是DbConnection）继承并且实现相同的接口（如IDbConnection）。这样的话，一旦你掌握了一种数据提供程序的用法，学习其他的数据提供程序就非常简单了。

说明 在ADO.NET下谈到“连接对象”时，大多指从DbConnection派生的类型，没有直接叫“Connection”的类。对于“命令对象”、“数据适配器对象”等也一样。按照命名惯例，特定数据提供程序中类的命名都以相关的DBMS名字作为前缀（如SqlConnection、OracleConnection和SqlDataReader）。

图21-2演示了ADO.NET数据提供程序的大致情况。注意，图中的客户程序集代表任何.NET应用程序：控制台程序、WindowsForms程序、WPF程序、ASP.NET网页、WCF服务、.NET类库等。

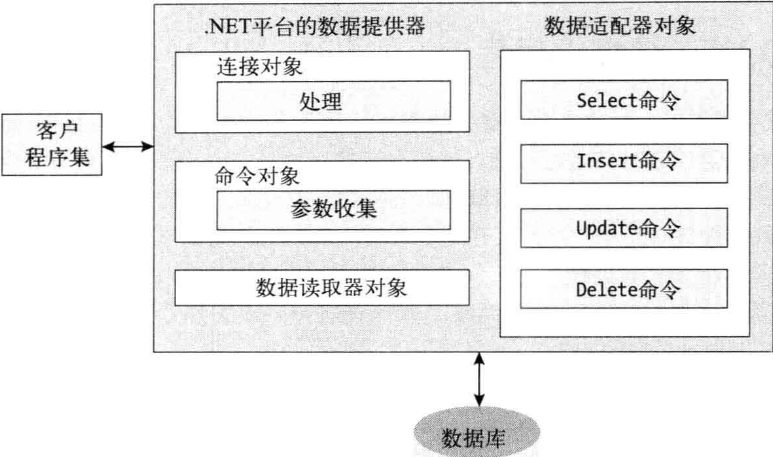


图21-2　ADO.NET数据提供程序提供了对特定DBMS的访问

肯定还有许多数据提供程序的类在图21-2中没有显示，但是所有的数据提供程序都有这些核心类。

21.2.1　微软提供的ADO.NET数据提供程序

微软的.NET提供了Oracle、SQL Server和OLE DB/ODBC系列等众多的数据提供程序。表21-2列举了每一个ADO.NET数据提供程序的命名空间和包含它们的程序集。

表21-2　微软ADO.NET数据提供程序

数据提供程序	命名空间	程　序　集
OLE DB	`System.Data.OleDb`	`System.Data.dll`
Microsoft SQL Server	`System.Data.SqlClient`	`System.Data.dll`
Microsoft SQL Server Mobile	`System.Data.SqlServerCe`	`System.Data.SqlServerCe.dll`
ODBC	`System.Data.Odbc`	`System.Data.dll`

说明　没有特定的数据提供程序直接映射到Jet引擎（比如微软Access数据库）。如果要和Access数据文件交互，可以使用OLE DB数据提供程序或者ODBC数据提供程序。

由定义在`System.Data.OleDb`命名空间下的类组成的OLE DB数据提供程序能让你访问所有支持基于传统COM的OLE DB协议的数据库。使用这个数据提供程序，能非常简单地改变连接字符串中的“Provider”，并能和各种OLE DB数据库进行通信。

需要知道的是，其实OLE DB数据提供程序在后台调用各种COM对象来实现数据交互，这可能会影响程序的性能。基本上当某个DBMS没有对应的.NET数据提供程序时，我们才会用OLE DB数据提供程序。然而，其实称职的DBMS都应该有自定义的ADO.NET数据提供程序提供下载，`System.Data.`

OleDb应该被认为是遗留命名空间，它在.NET 4.5的世界中用处很少（.NET 2.0下出现的数据提供程序工厂模型可能还会让它有点用，这会在稍后讨论）。

说明 有一种情况必须要使用System.Data.OleDb，那就是如果我们需要和微软SQL Server 6.5或更低的版本交互的话。System.Data.SqlClient命名空间只可以和微软SQL Server 7.0或更高的版本进行通信。

微软SQL Server数据提供程序提供了对SQL Server数据存储的直接访问（而且仅仅是7.0版本以及更高版本的SQL Server）。System.Data.SqlClient命名空间包含SQL Server数据提供程序的一些类，并且提供了和OLE DB数据提供程序差不多的一些功能，但是主要的区别是它绕开OLE DB层进行访问，带来的效率显而易见。同样，SQL Server数据提供程序也能很好利用DBMS的一些特性。

微软支持的其他提供程序（System.Data.Odbc和System.Data.SqlClientCe）提供了对ODBC连接以及SQL Server移动版本的交互性DBMS（通常用于诸如Windows Mobile之类的手持设备）的访问。通常只有当需要与没有自定义.NET数据提供程序的DBMS通信时，定义在System.Data.Odbc命名空间中的ODBC类型才有用，因为ODBC是对很多数据存储提供访问的通用模型。

21.2.2 关于System.Data.OracleClient.dll

.NET平台的较早版本包含System.Data.OracleClient.dll程序集。顾名思义，它是用来与Oracle数据库通信的数据提供程序。但在.NET 4.0中，该程序集被标记为过时的，并且最终将被弃用。

乍看上去，你可能会担心ADO.NET正在慢慢将所有注意力集中到微软的数据存储上来，但其实并不是这样。Oracle自己提供了一个自定义的.NET程序集，它与微软提供的数据提供程序遵循了同样的数据规范。如果要获得该.NET程序集，可以访问Oracle网站的下载区：

www.oracle.com/technology/tech/windows/odpnet/index.html

21.2.3 选择第三方的数据提供程序

除了微软自带的数据提供程序(和Oracle的自定义.NET类库)以外，还有很多针对各种开源和商业数据库的第三方数据提供程序。虽然我们更希望直接从数据库提供商这里获得ADO.NET的数据提供程序，但是也应该知道这个网站（请注意这个URL可能会改变）：

http://www.sqlsummit.com/DataProv.htm

这个网站列出了每一个已知的ADO.NET数据提供程序以及更多信息和下载的链接。在这里我们能找到许多ADO.NET提供程序，包括SQLite、IBMDB2、MySQL、PostgreSQL和Sybase等。

由于ADO.NET数据提供程序数量众多，本章中的示例会使用微软SQL Server数据提供程序（System.Data.SqlClient.dll）。回忆一下，这个提供程序允许我们和微软SQL Server 7.0及更高的版本进行交互，包括SQL Server Express版。如果希望使用ADO.NET和其他DBMS交互，那么在理解了后面介绍的内容后，应该就没什么问题了。

21.3　其他的 ADO.NET 命名空间

除了定义特定数据提供程序类型的一些.NET命名空间外，.NET基础类库同样还包含了一些另外的与ADO.NET相关的命名空间，如表21-3所示（同样，第23章会介绍与Entity框架相关的程序集和命名空间）。

表21-3　其他的ADO.NET相关的命名空间

命名空间	作　用
Microsoft.SqlServer.Server	这个命名空间提供的类型促进CLR和SQL Server 2005及后续版本的集成服务
System.Data	这个命名空间定义了各种数据提供程序所用的主要类型,包括公共接口和断开连接层的许多类型（如DataSet和DataTable等）
System.Data.Common	这个命名空间包含了各种数据提供程序共享的类型，也包括公共抽象基类
System.Data.Sql	这个命名空间能使你发现安装在当前本地网络的SQL Server的实例
System.Data.SqlTypes	这个命名空间包含微软SQL Server中使用的本机数据类型。尽管你可能不会直接使用相应的CLR数据类型，但是可以优化SqlTypes来和SQL Server交互（例如，如果SQLServer数据库包含整数值，你可以使用int或SqlTypes.SplInt32来表示它）

由于篇幅限制，本章不会研究每个ADO.NET命名空间中的每个类型。然而，非常有必要理解System.Data命名空间下的一些类型。

21.4　System.Data 命名空间的类型

在ADO.NET各种命名空间中，就属System.Data最常用了。如果不在数据访问应用程序里指定System.Data命名空间，基本上不能建立任何ADO.NET应用程序。这个命名空间包含了所有ADO.NET数据提供程序的一些不和基层数据库相关的公共类型。除了许多数据库方面的异常类型（如NoNullAllowedException、RowNotInTableException和MissingPrimaryKeyException等）外，System.Data还包含了各种数据库的基本类型（如表、行、列和约束等），同样它还包含了所有数据提供程序对象所实现的一些公共接口。表21-4列举了其中的一些核心类型。

表21-4　System.Data命名空间的核心成员

类　型	作　用
Constraint	表示某个DataColumn对象的约束
DataColumn	表示某个DataTable对象中的一列
DataRelation	表示两个DataTable对象之间的父子关系
DataRow	表示某个DataTable对象中的一行
DataSet	由多个相关DataTable对象组成的内存中的数据缓存
DataTable	表示内存数据的一个表
DataTableReader	使你能像fire-hose游标（只读向前）那样获取DataTable

（续）

类　型	作　用
DataView	表示用于排序、筛选、搜索、编辑和导航的DataTable的自定义视图
IDataAdapter	定义了数据适配器对象的主要行为
IDataParameter	定义了参数对象的主要行为
IDataReader	定义了数据读取器对象的主要行为
IDbCommand	定义了命令对象的主要行为
IDbDataAdapter	对IDataAdapter的一个扩展，增加了数据适配器对象的一些功能
IDbTransaction	定义了事务对象的主要行为

System.Data中的绝大多数类都在进行ADO.NET断开连接层编程时使用。在下一章中，我们会进而了解DataSet以及相关类型（如DataTable、DataRelation和DataRow等）的细节，并且了解如何使用它们（以及相关数据适配器）来表示和操作远程数据的客户端副本。

而接下来的任务是进一步了解System.Data的一些主要接口，这样能更好地理解数据提供程序的一些常用功能。整个章节都会贯穿介绍它们的细节，先来看一下每一个接口类型的主要行为。

21.4.1 IDbConnection接口的作用

首先是由数据提供程序的连接对象实现的IDbConnection类型，这个接口定义了一系列用于配置某个数据库连接的一些成员。通过它，你还能获取一个数据提供程序的事务对象。下面是这个接口的定义：

```
public interface IDbConnection : IDisposable
{
  string ConnectionString { get; set; }
  int ConnectionTimeout { get; }
  string Database { get; }
  ConnectionState State { get; }

  IDbTransaction BeginTransaction();
  IDbTransaction BeginTransaction(IsolationLevel il);
  void ChangeDatabase(string databaseName);
  void Close();
  IDbCommand CreateCommand();
  void Open();
}
```

说明　和.NET基础类库中的其他类型相似，调用Close()方法从功能上说和直接调用Dispose()方法或间接使用C# using作用域一样（见第13章）。

21.4.2 IDbTransaction接口的作用

我们发现IDbConnection接口定义的已重载的BeginTransaction()方法提供了一个数据提供程序的

事务对象。使用IDbTransaction接口定义的成员，你能够以编程方式在事务会话和基层的数据存储之间进行交互：

```
public interface IDbTransaction : IDisposable
{
  IDbConnection Connection { get; }
  IsolationLevel IsolationLevel { get; }

  void Commit();
  void Rollback();
}
```

21.4.3　IDbCommand接口的作用

接下来看一下由数据提供程序的命令对象实现的IDbCommand接口。和其他数据访问对象模型一样，命令对象让你能通过编程方式处理SQL语句、存储过程和参数化查询。另外，命令对象提供了已重载的ExecuteReader()方法来访问数据提供程序的数据读取器对象：

```
public interface IDbCommand : IDisposable
{
  string CommandText { get; set; }
  int CommandTimeout { get; set; }
  CommandType CommandType { get; set; }
  IDbConnection Connection { get; set; }
  IDataParameterCollection Parameters { get; }
  IDbTransaction Transaction { get; set; }
  UpdateRowSource UpdatedRowSource { get; set; }

  void Cancel();
  IDbDataParameter CreateParameter();
  int ExecuteNonQuery();
  IDataReader ExecuteReader();
  IDataReader ExecuteReader(CommandBehavior behavior);
  object ExecuteScalar();
  void Prepare();
}
```

21.4.4　IDbDataParameter和IDataParameter接口的作用

注意，IDbCommand的参数的Parameters属性返回的是实现IDataParameterCollection的强类型化的集合。这个接口提供了实现IDbDataParameter相关类型的访问（比如参数对象）：

```
public interface IDbDataParameter : IDataParameter
{
  byte Precision { get; set; }
  byte Scale { get; set; }
  int Size { get; set; }
}
```

IDbDataParameter接口扩展了IDataParameter接口来实现下列行为：

```
public interface IDataParameter
{
  DbType DbType { get; set; }
  ParameterDirection Direction { get; set; }
```

```
    bool IsNullable { get; }
    string ParameterName { get; set; }
    string SourceColumn { get; set; }
    DataRowVersion SourceVersion { get; set; }
    object Value { get; set; }
  }
```

我们知道，IDbDataParameter和IDataParameter接口的功能是让我们能通过ADO.NET的参数对象来表示SQL命令和存储过程中的参数，而不是把这些参数硬编码在字符串字面量内。

21.4.5 IDbDataAdapter和IDataAdapter接口的作用

数据适配器用来从特定的数据库获取和返回DataSet。基于此，IDbDataAdapter接口定义了如下的属性来保存实现相关的选择、插入、更新、删除操作的SQL语句：

```
public interface IDbDataAdapter : IDataAdapter
{
  IDbCommand DeleteCommand { get; set; }
  IDbCommand InsertCommand { get; set; }
  IDbCommand SelectCommand { get; set; }
  IDbCommand UpdateCommand { get; set; }
}
```

除了这4个属性以外，ADO.NET数据适配器也同样实现定义在父接口IDataAdapter中的一些功能。它定义了数据适配器类型的一些主要功能：使用Fill()和Update()方法在调用者和基层数据库之间传递DataSet。同样，也能使用IDataAdapter接口的TableMappings属性来实现数据库列的映射，使列名更加友好：

```
public interface IDataAdapter
{
  MissingMappingAction MissingMappingAction { get; set; }
  MissingSchemaAction MissingSchemaAction { get; set; }
  ITableMappingCollection TableMappings { get; }

  int Fill(DataSet dataSet);
  DataTable[] FillSchema(DataSet dataSet, SchemaType schemaType);
  IDataParameter[] GetFillParameters();
  int Update(DataSet dataSet);
}
```

21.4.6 IDataReader和IDataRecord接口的作用

下一个重要的接口是IDataReader，它定义了数据读取器对象的一些常用行为。当你从ADO.NET的数据提供程序获得一个数据读取器相关类型后，就能使用它以只读向前的形式循环提取数据：

```
public interface IDataReader : IDisposable, IDataRecord
{
  int Depth { get; }
  bool IsClosed { get; }
  int RecordsAffected { get; }

  void Close();
  DataTable GetSchemaTable();
  bool NextResult();
  bool Read();
}
```

最后我们看到，IDataReader扩展了IDataRecord，后者定义了一些成员让你能直接从流中获得强类型化的数据，而不是从数据读取器的重载索引器获得普通的System.Object对象后再去进行强制类型转换。下面列举了IDataRecord定义的部分GetXXX()方法（完整列表请参见.NET Framework 4.5 SDK）：

```
public interface IDataRecord
{
  int FieldCount { get; }
  object this[ string name ] { get; }
  object this[ int i ] { get; }
  bool GetBoolean(int i);
  byte GetByte(int i);
  char GetChar(int i);
  DateTime GetDateTime(int i);
  decimal GetDecimal(int i);
  float GetFloat(int i);
  short GetInt16(int i);
  int GetInt32(int i);
  long GetInt64(int i);
...
  bool IsDBNull(int i);
}
```

说明　IDataReader.IsDBNull()方法可以用于在从数据读取器获取值之前以编程方式查看某个字段是否被设置为null（来避免触发运行时异常）。回忆一下，C#支持可空数据类型（见第4章），与可空数据列交互时它会很有用。

21.5　使用接口的抽象数据提供程序

到这里我想你应该更了解.NET数据提供程序的一些常用功能了。前面说过，虽然各种数据提供程序的实现类型名字上不尽相同，但是我们能通过一种优雅的形式来对付这些相似的对象，即基于接口的多态。因此，如果你定义一个使用IDbConnection参数的方法，就能传递任何ADO.NET连接对象，如下所示：

```
public static void OpenConnection(IDbConnection cn)
{
  // 为调用者打开传递进来的连接对象
  cn.Open();
}
```

说明　接口不是必需的，使用抽象基类（如DbConnection）作为参数或返回值，也会得到相同效果。

对于返回值可以同样处理。例如，下面的C#控制台应用程序MyConnectionFactory实现了让调用者通过使用一个自定义的枚举来获得需要的连接对象，为了诊断，我们会通过反射服务来输出底层连接对象，然后输入以下代码：

```
using System;
...
// 需要这些来获取公共接口的定义以及各种用于测试的连接对象
```

```
using System.Data;
using System.Data.SqlClient;
using System.Data.Odbc;
using System.Data.OleDb;

namespace MyConnectionFactory
{
  // 可能的提供程序列表
  enum DataProvider
  { SqlServer, OleDb, Odbc, None }

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("**** Very Simple Connection Factory *****\n");

      // 获取某个连接
      IDbConnection myCn = GetConnection(DataProvider.SqlServer);
      Console.WriteLine("Your connection is a {0}", myCn.GetType().Name);

      // 打开、使用和关闭连接

      Console.ReadLine();
    }

    // 这个方法根据DataProvider枚举值返回某个连接对象
    static IDbConnection GetConnection(DataProvider dp)
    {
      IDbConnection conn = null;
      switch (dp)
      {
        case DataProvider.SqlServer:
          conn = new SqlConnection();
          break;
        case DataProvider.OleDb:
          conn = new OleDbConnection();
          break;
        case DataProvider.Odbc:
          conn = new OdbcConnection();
          break;
      }
      return conn;
    }
  }
}
```

通过使用System.Data下的这些接口（或System.Data.Common抽象基类），能够构建一个灵活的代码库，或许可以不用加班了。可能你现在在做一个针对SQL Server的应用程序，但是公司接下来要转向其他数据库平台了，怎么办呢？如果你的程序是使用System.Data.SqlClient进行硬编码的，并且后台数据库管理系统改变的话，你就不得不重新修改、编译、部署程序集。

使用应用程序配置文件增加灵活性

为进一步增加ADO.NET应用程序的灵活性，可以建立一个客户端*.config文件并且在<appSettings>

元素下使用一对自定义的键字/值的组合。在第14章中提到过，我们能通过System.Configuration命名空间内的一些类来以编程方式获取这些自定义数据。例如，假设你已经在配置文件内指定了一个数据提供程序值：

```
<configuration>
  <appSettings>
    <!-- 这个键值映射到枚举值中的某个值 -->
    <add key="provider" value="SqlServer"/>
  </appSettings>
</configuration>
```

这样，我们就可以更新Main()，以编程方式获得底层的数据提供程序。为此，我们必须构建一个连接对象工厂，以便在不重新编译代码库的情况下（只改变*.config文件）改变提供程序。下面是Main()的相关更新：

```
static void Main(string[] args)
{
  Console.WriteLine("**** Very Simple Connection Factory *****\n");

  // 读取提供程序键
  string dataProvString = ConfigurationManager.AppSettings["provider"];

  // 把字符串转换为枚举
  DataProvider dp = DataProvider.None;
  if(Enum.IsDefined(typeof(DataProvider), dataProvString))
    dp = (DataProvider)Enum.Parse(typeof(DataProvider), dataProvString);
  else
    Console.WriteLine("Sorry, no provider exists!");

  // 获取某个连接
  IDbConnection myCn = GetConnection(dp);
  if(myCn != null)
    Console.WriteLine("Your connection is a {0}", myCn.GetType().Name);

  // 打开、使用和关闭连接

  Console.ReadLine();
}
```

说明　要使用ConfigurationManager类型，请确保你已经引用了System.Configuration.dll程序集并且导入了System.Configuration命名空间。

至此，我们已经编写了一些ADO.NET代码以允许动态指定底层连接。然而，一个很明显的问题是，这个抽象只能在MyConnectionFactory.exe应用程序中使用。如果我们重新把这个示例转换成.NET代码类库（例如MyConnectionFactory.dll），就可以构建许多客户端来使用这个抽象层来获取各种连接对象。

然而，获取连接对象只是使用ADO.NET的一个方面。要做一个有用的数据提供程序工厂库，我们还必须处理命令对象、数据读取器、数据适配器、事务对象以及其他数据相关的类型。虽然构建这样一个代码类库不会很难，但是的确需要很多代码以及相当多的时间。

幸好.NET 2.0发布之后，Redmond的那些好人已经在.NET基础类库中直接构建了这个功能。我们稍后会研究这个正式的API，但是首先需要创建本章以及之后的很多章节都会使用的自定义数据库。

源代码 MyConnectionFactory项目的源代码位于Chapter 21子目录下。

21.6 创建 AutoLot 数据库

在本章中，我们会对一个叫AutoLot的简单SQL Server测试数据库执行查询。为了继续探讨本书中使用的汽车主题，这个数据库会包含三个相关表（Inventory、Orders和Customers），它们包含表示虚拟汽车销售公司订单的各种数据。

在这里我们假设你已经有微软SQL Server（7.0或更高版本）或微软SQL Server Express Edition。如果还没有，可以在这里下载：

www.microsoft.com/sqlserver/en/us/editions/2012-editions/express.aspx

这个轻量级的数据库服务器完全能满足我们的需要，一是因为免费，二是因为它们提供了GUI前端（SQL Server Management Tool）来创建和管理数据库，三是因为它可以和Visual Studio/Visual C# Express版本进行整合。

为了说明上面的问题，本节余下的内容会介绍使用Visual Studio构建AutoLot数据库。如果使用Visual C# Express，也可以通过使用Database Explorer窗口（可以从View→OtherWindows菜单项进行加载）执行这里提到的相似操作。

说明 要知道AutoLot数据库会在本书余下部分中一直用到。

21.6.1 创建Inventory表

现在就来构建测试数据库。首先启动Visual Studio，通过IDE的View菜单打开Server Explorer窗口。接着，右击Data Connection节点并且选择Create New SQL Server Database菜单项（如图21-3所示）。

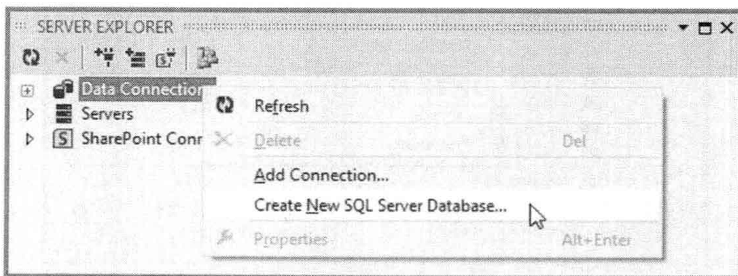


图21-3 使用Visual Studio创建新的SQL Server数据库

在出现的对话框内，你需要在Server name文本框中输入一个值，表示数据库要在那台机器上创建。如果你的机器上已经安装了SQL Server完整版本，输入(local)表示要在本机上创建数据库。但使用微软SQL Express的话，需要输入(local)\SQLEXPRESS。

将新数据库命名为AutoLot（选择Windows Authentication，如图21-4所示）。



图21-4　使用Visual Studio创建新的SQL Server Express数据库

至此，AutoLot数据库没有任何数据库对象（如表和存储过程等）。要插入新表，只需要右击Tables节点并且选择Add New Table项（如图21-5所示）。

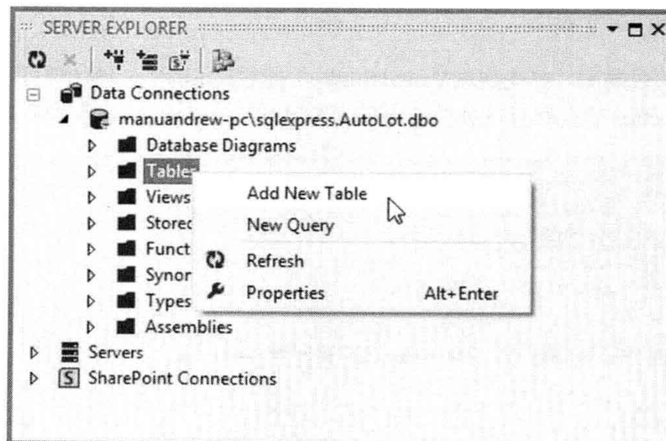


图21-5　添加Inventory表

使用表编辑器，添加4个数据列（CarID、Make、Color以及PetName），类型均为varchar(50)。确保CarID列被设置为主键（通过右击CarID行并且选择Set Primary Key）。此外，除了CarID之外所有列都可以设置为空值。图21-6显示了最后的表设置（不需要修改Column Properties编辑器中的任何项，但要注意每列的数据类型）。

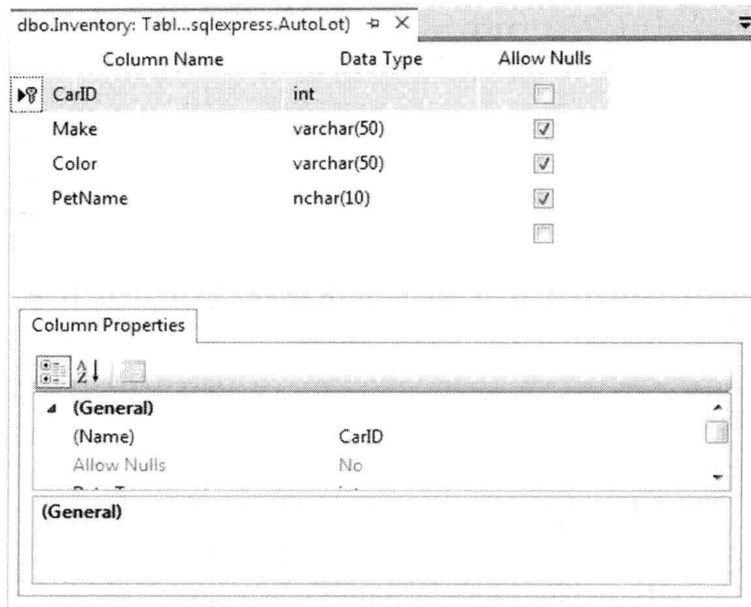


图21-6 设计Inventory表

创建了新的表架构之后，通过Ctrl+S或File→Save来保存工作。此时，系统会要求你为新表命名。在本例中，我们将表命名为Inventory。完成后应该可以在Server Explorer的Tables节点下看到Inventory表。

21.6.2 为Inventory表添加测试记录

要为第一个表格添加记录，右击Inventory表图标，选择Show Table Data。随便输入一个新的汽车（为了更有趣，请确保让一些汽车有相同的颜色和品牌）。图21-7显示了可能的库存列表。

Inventory: Query(m...sqlexpress.AutoLot)   Program.cs

CarID	Make	Color	PetName
32	VW	Black	Zippy
83	Ford	Rust	Rusty
872	Saab	Black	Mel
888	Yugo	Yellow	Clunker
1000	BMW	Black	Bimmer
1011	BMW	Green	Hank
2911	BMW	Pink	Pinky
NULL	NULL	NULL	NULL

4 of 7

图21-7 填充Inventory表

21.6.3　编写GetPetName()存储过程

在本章后面，我们会研究如何使用ADO.NET来调用存储过程。你可能已经知道，存储过程一般保存在某个数据库中，并且通常对表数据进行操作然后返回值。我们会增加一个存储过程，根据提供的CarID来返回某个汽车的昵称。只需要在Server Explorer中右击AutoLot数据库的Stored Procedures节点，然后选择Add New Stored Procedure就可以完成了。在编辑器中输入下面的代码：

```
CREATE PROCEDURE GetPetName
@carID int,
@petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarID = @carID
```

保存过程之后，它会根据CREATE PROCEDURE语句自动命名为GetPetName（注意第一次保存时，Visual Studio会自动将SQL脚本改为ALTER PROCEDURE）。然后，我们就可以在Server Explorer中看到新的存储过程了（如图21-8所示）。

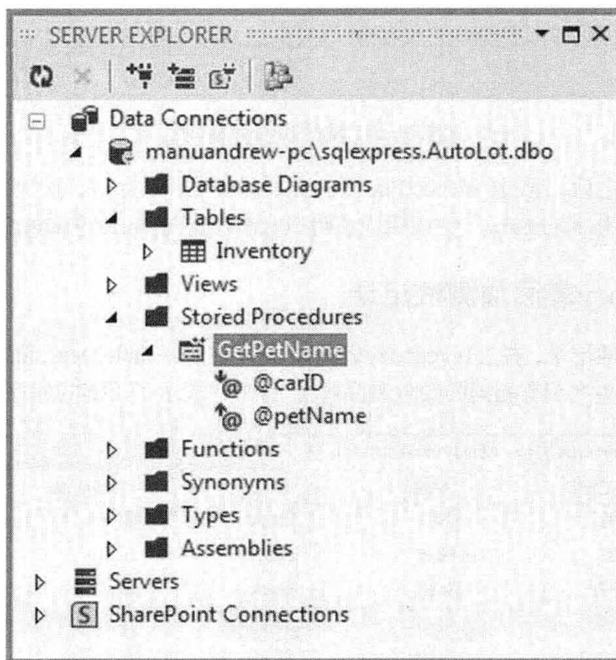


图21-8　GetPetName存储过程

说明　存储过程不是都需要像这里一样使用输出参数来返回数据，不过，这样做就为本章后面提到的SqlParameter的Direction属性做了铺垫。

21.6.4 创建Customers和Orders表

AutoLot数据库还会有另外两个表：Customers表和Orders表。Customers表会保存顾客列表，其中包括3列（应该被设置为主键的CustID，还有FirstName和LastName）。按照与创建Inventory表相同的步骤来创建如下结构的Customer表（如图21-9所示）。

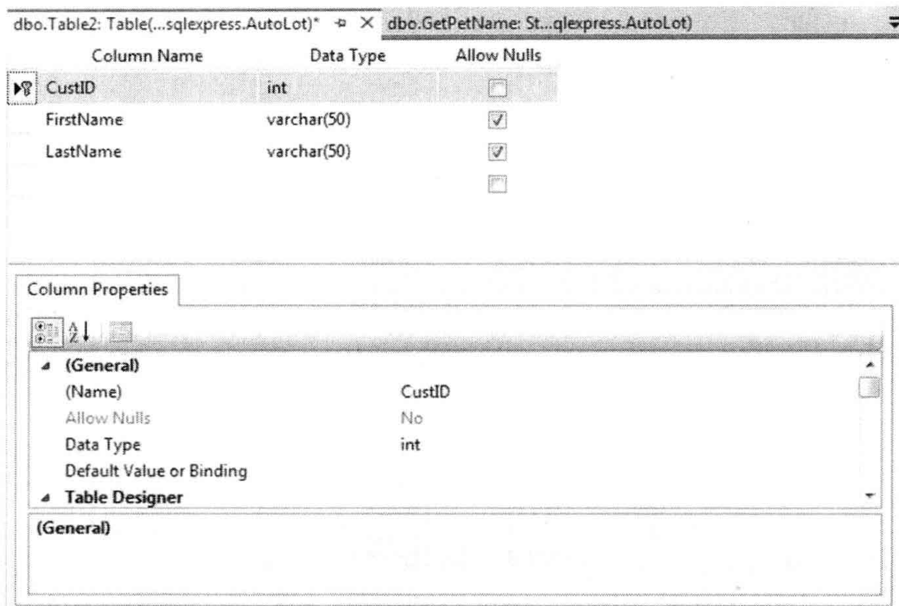


图21-9　设计Customers表

在为表命名并保存之后，增加一些顾客记录（如图21-10所示）。

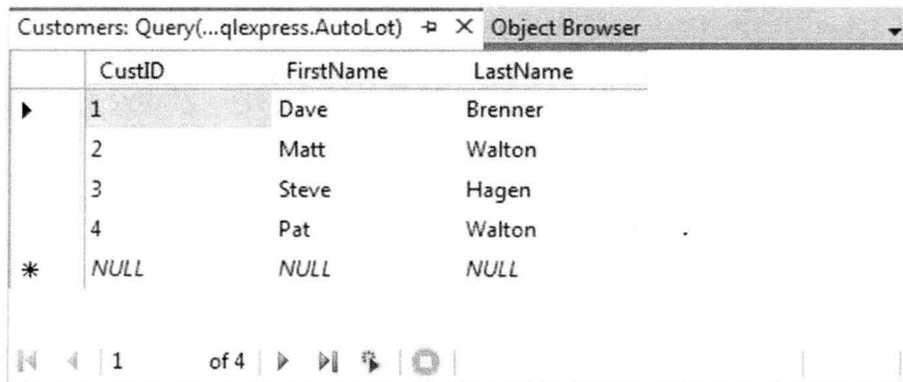


图21-10　填充Customers表

最后一个表Orders会用于表示某个顾客感兴趣的汽车，它把OrderID值映射到CarID/CustID值。图21-11显示了最后表的结构（再次注意，OrderID是主键）。

Column Name	Data Type	Allow Nulls
OrderID	int	
CustID	int	
CarID	int	

图21-11　设计Orders表

现在，为Orders表添加数据。假设OrderID值从1000开始，为每一个CustID值选择唯一的CarID（如图21-12所示）。

OrderID	CustID	CarID
1000	1	1000
1001	2	32
1002	3	888
1003	4	2911
NULL	NULL	NULL

图21-12　填充Orders表

我们可以看到，Dave Brenner（CustID=1）对黑色的BMW（CarID=1000）感兴趣，而Pat Walton（CustID=4）关注粉红色的BMW（CarID=2911）。

21.6.5 可视化创建表关系

最后一个任务就是在Customers、Orders以及Inventory表之间创建关系。使用Visual Studio来这么做很简单，因为我们只需在设计时插入新数据库图。这可以通过下面的方法实现：使用Server Explorer，右击AutoLot数据库的Database Diagram节点，选择Add New Diagram菜单项。然后，请确保在单击弹出对话框中的Add按钮之前选择所有的表（如图21-13所示）。

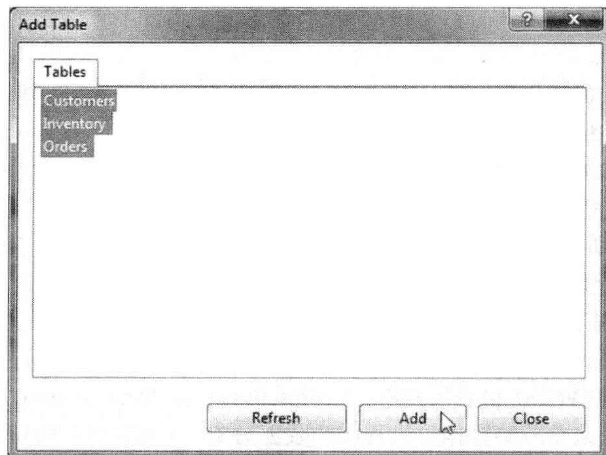


图21-13 选择表

要创建表之间的关系，首先单击Inventory表的CarID键，然后（保持按住鼠标按钮）拖放到Orders表的CarID字段。释放鼠标并接受弹出对话框的所有默认设定。现在，重复相同步骤来将Customers表的CustID键映射到Orders表的CustID字段。完成之后，我们可以看到图21-14中的类对话框。

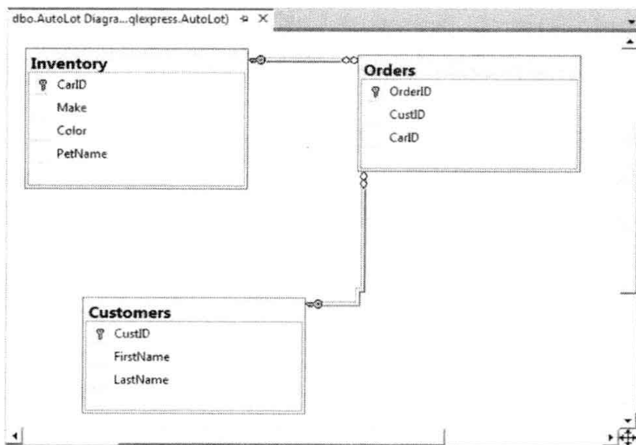


图21-14 关联的Orders、Inventory和Customers表

这样，AutoLot数据库就完成了！当然，这和真实的数据库差很远，不过用于本书余下部分就够了。既然已经有了可以用于测试的数据库，就让我们深入了解ADO.NET数据提供程序工厂模型的细节吧。

21.7　ADO.NET 数据提供程序工厂模型

.NET数据提供程序工厂模式能让我们用多种数据访问类型构建单个代码库。而且通过应用程序配置文件（和全新的<connectionStrings>片段），我们无需重新编译、重新部署程序集就能够更改和获取提供程序和连接字符串。

为了理解怎样实现数据提供程序工厂模式，回顾表21-1所说的，数据提供程序中的类都从相同的基类继承并且都被定义在System.Data.Common命名空间内。

- DbCommand：所有命令类的抽象基类。
- DbConnection：所有连接类的抽象基类。
- DbDataAdapter：所有数据适配器类的抽象基类。
- DbDataReader：所有数据读取器类的抽象基类。
- DbParameter：所有参数类的抽象基类。
- DbTransaction：所有事务类的抽象基类。

另外，每个微软提供的数据提供程序都有一个继承自System.Data.Common DbProviderFactory的类。这个基类定义了一些方法来获取某数据提供程序的数据对象。下面是DbProviderFactory相关成员的一段代码：

```
public abstract class DbProviderFactory
{
...
  public virtual DbCommand CreateCommand();
  public virtual DbCommandBuilder CreateCommandBuilder();
  public virtual DbConnection CreateConnection();
  public virtual DbConnectionStringBuilder CreateConnectionStringBuilder();
  public virtual DbDataAdapter CreateDataAdapter();
  public virtual DbDataSourceEnumerator CreateDataSourceEnumerator();
  public virtual DbParameter CreateParameter();
}
```

System.Data.Common命名空间下有一个叫做DbProviderFactories的类（注意这里Factory是复数）帮助我们从DbProviderFactory继承的类获取数据提供程序。使用静态方法GetFactory()，并指定包含提供程序功能的.NET命名空间的名称，能够获得代表该数据提供程序的DbProviderFactory对象，如下所示：

```
static void Main(string[] args)
{
  // 获取SQL数据提供程序工厂
  DbProviderFactory sqlFactory =
    DbProviderFactories.GetFactory("System.Data.SqlClient");
...
}
```

读者可能会想，为什么要把提供程序的类型用字符串硬编码而不是直接从.config文件读取这些信

息呢（就像前面的MyConnectionFactory示例）？你仅需修改少量代码就能实现，不管怎么样，只要得到了某个数据提供程序的DbProviderFactory以后，你就能获取它相关的一些数据对象（比如连接、命令、数据读取器等）。

说明 实际上，可以把数据提供程序所在的.NET命名空间的名称作为参数传递给DbProviderFactories.GetFactory()。machine.config使用该字符串值从全局程序集缓存中动态加载正确的库。

21.7.1 完整的数据提供程序工厂的例子

作为完整的示例，让我们新建一个C#控制台应用程序DataProviderFactory来输出AutoLot数据库的汽车库存。对于这个初始的示例，我们会直接在DataProviderFactory.exe中硬编码数据访问逻辑（只是确保现在做的东西足够简单）。然而，在我们开始深入介绍ADO.NET编程模型之后，要将数据逻辑隔离到某个.NET代码类库中，本书余下部分都将会用到。

首先，增加System.Configuration.dll程序集的引用并且导入System.Configuration命名空间。然后，将App.config文件插入到当前项目中，并定义一个空的<appSettings>元素。新增一个叫provider的键，映射到我们希望获取的数据提供程序的命名空间（System.Data.SqlClient）。同样，在SQL Server Express的本地实例上定义一个表示到AutoLot数据库的连接的连接字符串：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <appSettings>
  <!-- 哪个提供程序 -->
  <add key="provider" value="System.Data.SqlClient" />

  <!-- 哪个连接字符串 -->
  <add key="cnStr" value= "Data Source=(local)\SQLEXPRESS;
          Initial Catalog=AutoLot;Integrated Security=True"/>
 </appSettings>
</configuration>
```

说明 稍后我们会详细研究连接字符串。然而，要知道如果我们在Server Explorer中选择AutoLot数据库图标，就可以从Visual Studio Properties窗口的Connection String属性中复制并粘贴正确的连接字符串。

现在你已经建立了一个正确的*.config文件，可以通过ConfigurationManager.AppSetting索引器读取provider和cnStr的值。provider的值被传到DbProviderFactories.GetFactory()方法来获取某个数据提供程序的工厂类型，同样cnStr的值会被继承自DbConnecion的对象用做ConnectionString属性。

假设你已经使用了System.Data和System.Data.Common命名空间，然后按照下面的代码来修改Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Data Provider Factories *****\n");
```

```
// 从*.config文件获取连接字符串和提供程序
string dp =
  ConfigurationManager.AppSettings["provider"];
string cnStr =
  ConfigurationManager.AppSettings["cnStr"];

// 得到工厂提供程序
DbProviderFactory df = DbProviderFactories.GetFactory(dp);

// 得到连接对象
using (DbConnection cn = df.CreateConnection())
{
  Console.WriteLine("Your connection object is a: {0}", cn.GetType().Name);
  cn.ConnectionString = cnStr;
  cn.Open();

  // 得到命令对象
  DbCommand cmd = df.CreateCommand();
  Console.WriteLine("Your command object is a: {0}", cmd.GetType().Name);
  cmd.Connection = cn;
  cmd.CommandText = "Select * From Inventory";

  // 从数据读取器输出数据
  using (DbDataReader dr = cmd.ExecuteReader())
  {
    Console.WriteLine("Your data reader object is a: {0}", dr.GetType().Name);

    Console.WriteLine("\n***** Current Inventory *****");
    while (dr.Read())
      Console.WriteLine("-> Car #{0} is a {1}.",
        dr["CarID"], dr["Make"].ToString());
  }
}
Console.ReadLine();
}
```

值得注意的是，为了更好地了解内幕，我们通过反射服务输出了基层连接、命令和数据读取器对象的全名。运行程序后，你能发现AutoLot数据库的Inventory表中的当前数据输出到了控制台。

```
***** Fun with Data Provider Factories *****

Your connection object is a: SqlConnection
Your command object is a: SqlCommand
Your data reader object is a: SqlDataReader

***** Current Inventory *****
-> Car #32 is a VW.
-> Car #83 is a Ford.
-> Car #872 is a Saab.
-> Car #888 is a Yugo.
-> Car #1000 is a BMW.
-> Car #1011 is a BMW.
-> Car #2911 is a BMW.
```

现在，如果修改*.config文件指定System.Data.OldDb作为数据提供程序（并且更新连接字符串的

Provider部分），如下所示：

```
<configuration>
 <appSettings>
  <!-- 哪个提供程序 -->
  <add key="provider" value="System.Data.OleDb" />

  <!-- 哪个连接字符串 -->
  <add key="cnStr" value=
  "Provider=SQLOLEDB;Data Source=(local)\SQLEXPRESS;
     Integrated Security=SSPI;Initial Catalog=AutoLot"/>
 </appSettings>
</configuration>
```

这代表后台使用了System.Data.OleDb类型，并产生如下结果：

```
***** Fun with Data Provider Factories *****

Your connection object is a: OleDbConnection
Your command object is a: OleDbCommand
Your data reader object is a: OleDbDataReader

***** Current Inventory *****
-> Car #32 is a VW.
-> Car #83 is a Ford.
-> Car #872 is a Saab.
-> Car #888 is a Yugo.
-> Car #1000 is a BMW.
-> Car #1011 is a BMW.
-> Car #2911 is a BMW.
```

当然，看到这里你可能还不是很清楚连接、命令和数据读取器对象分别起什么作用，你现在不需要关注过多的细节（后面还有很多篇幅会介绍）。通过这个例子，你只需要明白在ADO.NET下，以声明方式可以构建能适应多种数据提供程序的单个代码库。

21.7.2 数据提供程序工厂模型的潜在缺陷

尽管这个模型很强，但是你要知道，代码库其实只能通过抽象基类的成员来使用所有提供程序通用的一些类型和方法。因此在写代码库的时候，你会被局限于System.Data.Common命名空间下的DbConnection、DbCommand等其他类型公开的成员。

这样，你会发现这种“泛化的”方式使得我们不能直接访问特定DBMS的漂亮功能。如果我们必须调用基础提供程序（比如SqlConnection）的特殊成员，那么可以通过显式强制类型转换就能实现，例如：

```
using (DbConnection cn = df.CreateConnection())
{
  Console.WriteLine("Your connection object is a: {0}", cn.GetType().Name);
  cn.ConnectionString = cnStr;
  cn.Open();
  if (cn is SqlConnection)
  {
    // 输出使用的SQL Server版本
```

```
        Console.WriteLine(((SqlConnection)cn).ServerVersion);
      }
    ...
    }
```

然而，这样会使代码变得不易维护（也缺乏灵活性），因为我们必须增加许多运行时检测。但如果你希望以最灵活的方式构建数据访问库，数据提供程序工厂模型提供了一个很好的机制。

21.7.3 <connectionStrings>元素

我们的连接字符串数据现在在*.config文件的<appSettings>元素中，应用程序配置文件定义了一个新的元素，叫做<connectionStrings>。你能在这个元素内设置任意多的名称/值组合并且能通过ConfigurationManager.ConnectionStrings索引器以编程方式访问。这种方式的（相比使用<appSettings>元素和ConfigurationManager.AppSettings索引器来访问）好处在于它能为你的应用程序以一个统一的方式定义多个连接字符串。

做一个实验，请按照下面的代码修改App.config文件（注意每一个连接字符串使用name和connectionString特性而不是key和value特性）：

```
<configuration>
 <appSettings>
  <!-- 哪个提供程序 -->
  <add key="provider" value="System.Data.SqlClient" />
 </appSettings>

 <!-- 这里是连接字符串 -->
 <connectionStrings>
  <add name ="AutoLotSqlProvider" connectionString =
       "Data Source=(local)\SQLEXPRESS;
        Integrated Security=SSPI;Initial Catalog=AutoLot"/>

  <add name ="AutoLotOleDbProvider" connectionString =
       "Provider=SQLOLEDB;Data Source=(local)\SQLEXPRESS;
        Integrated Security=SSPI;Initial Catalog=AutoLot"/>
 </connectionStrings>
</configuration>
```

现在你还需要修改Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Data Provider Factories *****\n");
  string dp =
    ConfigurationManager.AppSettings["provider"];
  string cnStr =
    ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].ConnectionString;
...
}
```

至此，我们已经有了一个应用程序可以显示AutoLot数据库中Inventory表的结果。我们看到，通过把提供程序名字和连接字符串放到外部*.config文件，数据提供程序工厂模型可以在背后动态加载合适的提供程序。有了这第一个示例，我们就可以深入了解ADO.NET连接层的更多细节。

说明 现在你已经理解ADO.NET数据提供程序工厂的作用，本书余下的例子都将直接使用System.Data.SqlClient命名空间中的类型，这样我们能更多关注手头讨论的东西。如果使用的是另外的DBMS（如Oracle），就需要相应地修改代码库。

源代码 DataProviderFactory项目的源代码位于Chapter 21子目录下。

21.8 ADO.NET 的连接层

前面曾提到过，ADO.NET的*连接层*允许通过数据提供程序的连接、命令、数据读取器对象与数据库进行交互。尽管在前面的DataProviderFactory例子里已经用过这些对象，我们还是再来看一下整个流程。当想连接数据库并且使用一个数据读取器对象来读取数据时，需要实现下面的几个步骤。

- 创建、配置、打开连接对象。
- 创建、配置一个命令对象，通过构造参数或Connection属性指定连接对象。
- 执行配置后的命令对象的ExecuteReader()方法。
- 使用数据读取器的Read()方法一条一条处理记录。

要熟练掌握这些对象，我们建立一个新的控制台应用程序AutoLotDataReader，并使用System.Data和System.Data.SqlClient命名空间。下面是Main()函数的完整代码和相关分析：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Data Readers *****\n");

    // 建立并打开一个连接
    using(SqlConnection cn = new SqlConnection())
    {
      cn.ConnectionString =
        @"Data Source=(local)\SQLEXPRESS;Integrated Security=SSPI;" +
         "Initial Catalog=AutoLot";
      cn.Open();

      // 建立一个SQL命令对象
      string strSQL = "Select * From Inventory";
      SqlCommand myCommand = new SqlCommand(strSQL, cn);

      // 通过ExecuteReader()获取一个数据读取器
      using(SqlDataReader myDataReader = myCommand.ExecuteReader())
      {
        // 循环所有的记录
        while (myDataReader.Read())
        {
          Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
                            myDataReader["Make"].ToString(),
                            myDataReader["PetName"].ToString(),
                            myDataReader["Color"].ToString());
        }
```

```
            }
        }
        Console.ReadLine();
    }
}
```

21.8.1　使用连接对象

使用数据提供程序需要做的第一步就是使用连接对象（前面说过，从DbConnection继承）去和数据库建立一个会话。.NET连接对象需要靠一个符合一定格式的连接字符串进行创建，通常它包含了几对名称/值的信息并且通过分号分割。这些信息用来确定你想要连接的计算机名、必要的安全配置、机器上数据库的名字和其他一些数据提供程序特有的信息。

从之前的代码中我们可以推断，Initial Catalog名涉及我们需要建立会话的数据库。Data Source名表示维护数据库的机器名。在这里，(local)允许我们定义一个标识来指示当前本地机器（不管这个机器的名字是什么），而\SQLEXPRESS标识通知SQL Server提供程序：正在连接到SQL Server Express默认版本的安装。（如果我们在完整版本的SQL Server上创建AutoLot，就只需要写Data Source=(local)。）

除了这些，我们还可以提供许多表示安全资格的标识。在这里，我们设置Integrated Security为SSPI（等价于true），它使用当前的Windows账户凭据作为用户身份验证。

说明　你可以查阅.NET Framework 4.5 SDK中数据提供程序的连接对象的ConnectionString属性，了解每一个DBMS的名称/值组合的细节。

在传入了连接字符串后，就可以调用Open()来建立和DBMS之间的连接。除了ConnectionString、Open()和Close()成员外，连接对象还提供了一些成员让你能进行一些额外的配置，比如说超时设置和事务信息。表21-5列举了一些（但不是全部）DbConnection基类的成员。

表21-5　DbConnection类型的成员

成　　员	作　　用
BeginTransaction()	用来开始数据库事务
ChangeDatabase()	为打开的连接更改当前数据库
ConnectionTimeout	这个只读属性返回建立连接时终止尝试并生成错误之前所等待的时间（默认15秒）如果想修改这个值，请在连接字符串中加入Connect Timeout片段（比如Connect Timeout=30）
Database	获取连接对象的数据库名
DataSource	获取连接对象的数据库服务器名
GetSchema()	返回一个包含数据源结构信息的DataTable对象
State	这个只读属性获取当前连接状态，表示为ConnectionState枚举形式

我们知道，DbConnection类型的这些属性实际上是只读的，只有我们想在运行时得到连接的一些特征时才会去用它们。如果需要修改这些默认设置的话，必须改变连接字符串本身。比如，设置连接字符串把超时时间从默认的15秒改为30秒：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Data Readers *****\n");

  using(SqlConnection cn = new SqlConnection())
  {
    cn.ConnectionString =
      @"Data Source=(local)\SQLEXPRESS;" +
      "Integrated Security=SSPI;Initial Catalog=AutoLot;Connect Timeout=30";
    cn.Open();

  // 新的辅助方法（见下面）
  ShowConnectionStatus(cn);
...
}
```


在前面的代码中我们注意到，这里把连接对象当做参数传入到了Program类的一个叫做Show-ConnectionStatus()的新辅助方法中，方法实现如下：

```
static void ShowConnectionStatus(SqlConnection cn)
{
  // 显示当前连接对象的各种状态
  Console.WriteLine("***** Info about your connection *****");
  Console.WriteLine("Database location: {0}", cn.DataSource);
  Console.WriteLine("Database name: {0}", cn.Database);
  Console.WriteLine("Timeout: {0}", cn.ConnectionTimeout);
  Console.WriteLine("Connection state: {0}\n", cn.State.ToString());
}
```

虽然大多数属性看字面都能知道它们的作用，但是State属性还是值得特别提一下。尽管它的值是ConnectionState这个枚举定义的，如下所示：

```
public enum ConnectionState
{
  Broken, Closed,
  Connecting, Executing,
  Fetching, Open
}
```

但是ConnectionState中只有ConnectionState.Open和ConnectionState.Close是有效的（枚举其他的成员为将来的使用所保留）。还有，如果当前的连接状态是ConnectionState.Close，再去关闭一次连接不会出现什么问题。

21.8.2 使用ConnectionStringBuilder对象

以编程方式使用连接字符串是非常有用的，因为如果经常用一个文本字符串来写连接字符串的话，不易于维护而且容易出问题。微软的ADO.NET数据提供程序现在提供了连接字符串构造函数对象，通过它能使用强类型化的属性来指定名称/值对。按照下面的例子修改当前的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Data Readers *****\n");

  // 通过构造函数对象来建立连接字符串
  SqlConnectionStringBuilder cnStrBuilder =
    new SqlConnectionStringBuilder();
```

```
    cnStrBuilder.InitialCatalog = "AutoLot";
    cnStrBuilder.DataSource = @"(local)\SQLEXPRESS";
    cnStrBuilder.ConnectTimeout = 30;
    cnStrBuilder.IntegratedSecurity = true;

    using(SqlConnection cn = new SqlConnection())
    {
      cn.ConnectionString = cnStrBuilder.ConnectionString;
      cn.Open();

      ShowConnectionStatus(cn);
  ...
    }
    Console.ReadLine();
  }
```

在这个例子中，首先建立了SqlConnectionStringBuilders的实例并且逐一设置属性，然后通过ConnectionString属性获取内部的连接字符串。其实还可以使用ConnectionStringBuilder的构造方法，只需要先建立一个连接字符串然后把它作为构造参数传入数据提供程序的连接字符串构造函数即可（如果想从App.config文件动态读取这些值，这个功能就非常有用）。一旦传入了连接字符串，你就能通过相关的属性来修改名称/值对，例如：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Data Readers *****\n");

  // 假设实际上是用*.config文件获取的cnStr
  string cnStr = @"Data Source=(local)\SQLEXPRESS;" +
                 "Integrated Security=SSPI;Initial Catalog=AutoLot";

  SqlConnectionStringBuilder cnStrBuilder =
    new SqlConnectionStringBuilder(cnStr);

  // 改变timeout的值
  cnStrBuilder.ConnectTimeout = 5;
...
}
```

21.8.3　使用命令对象

现在你对连接对象的作用更加清楚了，接下来我们就来讨论如何提交SQL查询到数据库。SqlCommand类型（继承自DbCommand）是SQL查询、表名和存储过程的一种面向对象的表示方法。可以用CommandType属性来指定命令的类型，它的值由CommandType枚举定义，如下所示：

```
public enum CommandType
{
  StoredProcedure,
  TableDirect,
  Text  // 默认值
}
```

当创建一个命令对象的时候，能通过传入构造参数或者指定CommandText属性来建立SQL查询。当然，还需要指定一个将要使用的连接对象。同样，连接对象也可以通过传入构造参数或者通过Connection属性来指定。看一下下面的代码段：

```
// 通过ctor参数建立命令对象
string strSQL = "Select * From Inventory";
SqlCommand myCommand = new SqlCommand(strSQL, cn);

// 通过属性建立命令对象
SqlCommand testCommand = new SqlCommand();
testCommand.Connection = cn;
testCommand.CommandText = strSQL;
```

直到这一步，其实你还没有真正提交SQL查询到AutoLot数据库，而只是建立了将要使用的命令对象状态。表21-6列举了一些DbCommand类型的其他成员。

表21-6 DbCommand类型的成员

成　员	作　用
CommandTimeout	获取或设置在终止执行命令的尝试并生成错误之前的等待时间，默认30秒
Connection	获取或设置此DbCommand实例使用的DbConnection
Parameters	获取DbParameter对象的集合，用于参数化查询
Cancel()	取消执行命令
ExecuteReader()	执行SQL查询并返回一个数据提供程序的DbDataReader对象，以只读向前的方式访问查询结果
ExecuteNonQuery()	提交SQL语句到数据库，不会有返回值（如插入、更新、删除或创建表）
ExecuteScalar()	一个轻量级版本的ExecuteReader()，用于返回一个值的查询（比如得到记录总数）
Prepare()	在数据源上创建该命令的准备好（或已编译）的版本，我们知道这样一个查询的执行速度会稍微快些（特别是在你想多次执行相同查询的情况下）

21.9 使用数据读取器

建立了活动的连接和SQL命令以后，下一步就是向数据源提交查询。你可能也猜到了这有很多方法能实现。但DbDataReader类型（实现IDataReader接口）是从数据源获取信息最简单也最快速的方法。前面提到过，数据读取器是只读向前的数据流，并且一次返回一条记录。因此，只有当你向数据源提交Select查询语句的时候，数据读取器才有用。

当你需要快速获取大批数据并且不需要在内存中存储它们的时候，数据读取器就非常有用了。比如，你想从表中取出20 000条记录保存到文本文件，先用DataSet保存这些数据到内存中就非常不值得了（因为DataSet同时在内存中保存整个结果）。

一个更好的方式是用数据读取器快速遍历每条记录。需要知道的是，数据读取器对象（不像数据适配器对象，后面会讨论它）会保持打开的连接，除非显式地关闭会话。

通过执行命令对象的ExecuteReader()方法来获得数据读取器对象。数据读取器（data reader）表示从数据库中读取的当前记录。它包含一个索引器方法（如C#中的[]语法），可以通过名称或从0开始的整数来访问当前记录中的列。

下面的例子表明，可以通过Read()方法的返回值来确定什么时候到达了最后一条记录（返回false）。对于每一条记录，我们使用索引器来输出每一个汽车的商标、昵称、颜色的值。注意，当结束操作记录后请尽快执行数据读取器的Close()方法，以释放连接对象：

```
static void Main(string[] args)
{
...
  // 通过ExecuteReader()得到一个数据读取器对象
  using(SqlDataReader myDataReader = myCommand.ExecuteReader())
  {
    // 循环所有记录
    while (myDataReader.Read())
    {
      Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
        myDataReader["Make"].ToString(),
        myDataReader["PetName"].ToString(),
        myDataReader["Color"].ToString());
    }
  }
  Console.ReadLine();
}
```

在前面的代码段中，数据读取器对象的索引器已经重载，可以使用string（表示列名）或者int（表示列的原始位置）两种方式来取值。因此，可以像下面一样改写（避免了硬编码字符串名）前面的代码（注意FieldCount属性的使用）：

```
while (myDataReader.Read())
{
  Console.WriteLine("***** Record *****");
  for (int i = 0; i < myDataReader.FieldCount; i++)
  {
    Console.WriteLine("{0} = {1} ",
      myDataReader.GetName(i),
      myDataReader.GetValue(i).ToString());
  }
  Console.WriteLine();
}
```

编译运行这个项目后，我们会看到程序列出了AutoLot数据库Inventory表中所有汽车的记录。下面的输出结果显示来自我的AutoLot数据库的几条记录：

```
***** Fun with Data Readers *****

***** Info about your connection *****
Database location: (local)\SQLEXPRESS
Database name: AutoLot
Timeout: 30
Connection state: Open

***** Record *****
CarID = 83
Make = Ford
Color = Rust
PetName = Rusty

***** Record *****
CarID = 107
Make = Ford
Color = Red
PetName = Snake
```

使用数据读取器获取多个结果集

我们能使用数据读取器对象从一个命令对象获取多个结果集。比如说，你要获取Inventory表和Customers表的所有行，要写两个SQL Select语句并且用分号分割，如下所示：

```
string strSQL = "Select * From Inventory;Select * from Customers";
```

获得了数据读取器对象后，就能通过NextResult()方法循环所有的结果集。需要知道的是，它会总是自动返回第一个结果集。因此，当需要遍历每一个表的行的时候，可以像下面一样建立循环语句：

```
do
{
  while (myDataReader.Read())
  {
    Console.WriteLine("***** Record *****");
    for (int i = 0; i < myDataReader.FieldCount; i++)
    {
      Console.WriteLine("{0} = {1}",
        myDataReader.GetName(i),
        myDataReader.GetValue(i).ToString());
    }
    Console.WriteLine();
  }
} while (myDataReader.NextResult());
```

至此，读者应该对数据读取器对象访问数据表的功能更为清楚了。要始终记住，数据读取器只能处理SQL的Select语句，不能通过Insert、Update或Delete请求来修改既有数据库表。要理解如何修改既有数据库，需要对命令对象进行进一步研究。

源代码 AutoLotDataReader项目的源代码位于Chapter 21子目录下。

21.10 构建可重用的数据访问库

通过前面的章节我们看到了，ExecuteReader()方法得到的数据读取器对象允许执行SQL Select语句并且以只读向前的形式获取信息。然而，当想向某表提交SQL语句来实现修改的时候（或其他非查询的SQL语句，如创建表或授权），就要用到命令对象的ExecuteNonQuery()方法了。这个方法能根据命令文本的格式执行相应的插入、修改和删除操作。

说明 严格来说，“非查询”是指没有返回结果集的SQL语句。因此，Select语句是查询，而Insert、Update和Delete语句则不是。因此，ExecuteNonQuery()返回的int值代表被影响的行数，而不是一组新记录。

为了阐明如何只通过调用ExecuteNonQuery()来修改既有数据库，下一个目标就是构建一个自定义的数据访问类库来封装对AutoLot数据库的操作过程。在生产级别的环境中，ADO.NET逻辑总是应该

隔离到.NET的一个*.dll程序集中，原因很简单：代码重用！本章第一个示例没有这么做，是为了让我们手头的事情变得简单。然而，我们可以想象，对于需要和AutoLot数据库交互的应用程序来说，需要编写相同的连接逻辑、相同的读逻辑以及相同的命令逻辑，这会浪费很多时间。

通过把数据访问逻辑隔离到.NET代码库，多个使用任何前端（基于控制台的、基于桌面的或者基于Web的，等等）的应用程序都可以以语言独立的方式直接引用类库。因此，如果使用C#编写数据类库，其他开发者就可以使用他们喜欢的.NET语言来构建UI。

在本章中，数据类库AutoLotDAL.dll会包含一个命名空间AutoLotConnectedLayer，它使用ADO.NET的连接类型和AutoLot进行交互。在下一章中，我们会新增一个命名空间（AutoLot DisconnectionLayer）到相同的*.dll中，其中包含的类型使用断开连接层和AutoLot进行通信。在本书余下的内容中，将有很多应用程序会用到这个类库。

首先，新建一个叫AutoLotDAL（AutoLot Data Access Layer的缩写）的C#类库项目，并且重命名初始C#代码文件为AutoLotConnDAL.cs。然后，重命名我们的命名空间域为AutoLotConnectedLayer，并且改变初始类的名字为InventoryDAL，因为这个类会定义各种成员，与AutoLot数据库的Inventoty表进行交互。最后，导入如下.NET命名空间：

```
using System;
…

// 我们会使用SQL server提供程序，然而，使用ADO.NET工厂模式来获得更好的灵活性也是可以的
using System.Data;
using System.Data.SqlClient;

namespace AutoLotConnectedLayer
{
  public class InventoryDAL
  {
  }
}
```

说明　从第13章中知道，如果对象使用管理底层资源（如数据库连接）的类型，实现IDisposable以及编写正确的终结器就是最佳实践。在生产环境中，诸如InventoryDAL的类也应该一样，但我们重点关注的是ADO.NET，所以我们没有这么做。

21.10.1　增加连接逻辑

我们要做的第一件事情就是定义一些方法允许调用者使用有效的连接字符串连接到某个数据源或从数据源断开。因为AutoLotDAL.dll程序集会硬编码使用System.Data.SqlClient类型，所以要定义一个在创建InventoryDAL对象时分配的SqlConnection私有变量。同样，要定义OpenConnection()方法和CloseConnection()方法来和这个成员变量进行交互，如下所示：

```
public class InventoryDAL
{
  // 这个成员会被所有方法使用
  private SqlConnection sqlCn = null;
```

```
    public void OpenConnection(string connectionString)
    {
      sqlCn = new SqlConnection();
      sqlCn.ConnectionString = connectionString;
      sqlCn.Open();
    }

    public void CloseConnection()
    {
      sqlCn.Close();
    }
  }
```

简单起见，InventoryDAL类型不会检测可能的异常，也不会在各种情况下抛出自定义异常（比如不正确的连接字符串）。如果要构建一个工业标准的数据访问库，最好为各种运行时异常情况使用结构化异常处理技术。

21.10.2 增加插入逻辑

增加新记录到Inventory表就像格式化SQL Insert语句（根据用户输入）和使用命令对象调用ExecuteNonQuery()这么简单。例如，增加一个公共方法InsertAuto()到InventoryDAL类型中，这个方法接受4个参数，它们分别对应Inventory表的4个列（CarID、Color、Make和PetName）。使用这些参数，格式化一个字符串类型来插入新记录。最后，使用SqlConnection对象，执行SQL语句：

```
public void InsertAuto(int id, string color, string make, string petName)
{
  // 格式化并且执行SQL语句
  string sql = string.Format("Insert Into Inventory" +
    "(CarID, Make, Color, PetName) Values" +
    "('{0}', '{1}', '{2}', '{3}')", id, make, color, petName);

  // 使用我们的连接来执行
  using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    cmd.ExecuteNonQuery();
  }
}
```

这个方法在语法上没有问题，但你可以提供一个重载版本，允许调用者传递一个表示行数据的强类型类。定义如下所示的NewCar类，表示Inventory表的新行：

```
public class NewCar
{
  public int CarID { get; set; }
  public string Color { get; set; }
  public string Make { get; set; }
  public string PetName { get; set; }
}
```

现在，在InventoryDAL类中添加下面这个版本的InsertAuto()：

```
public void InsertAuto(NewCar car)
{
  // 格式化并且执行SQL语句
  string sql = string.Format("Insert Into Inventory" +
    "(CarID, Make, Color, PetName) Values" +
```

```
    "('{0}', '{1}', '{2}', '{3}')", car.CarID, car.Make, car.Color, car.PetName);

  // 使用我们的连接来执行
  using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    cmd.ExecuteNonQuery();
  }
}
```

定义表示关系数据库中记录的类，是构建数据访问库的常见方法。事实上，你将在第23章中看到，ADO.NET Entity Framework自动生成强类型的类，来与数据库中的数据进行交互。而ADO.NET的非连接层（见第22章）生成强类型的DataSet对象，表示关系型数据库某个表中的数据。

说明 你可能知道，使用字符串拼接构建SQL语句可能会有安全风险（比如SQL注入攻击）。推荐使用参数化查询来构建命令文本，稍后我们会探讨。

21.10.3 增加删除逻辑

删除既有记录和插入新记录一样简单。和InsertAuto()的代码清单不同，我会演示如何使用重要的try/catch块域来处理可能试图删除一辆当前已被Customers表中的一位顾客下了订单的汽车。为InventoryDAL类类型增加如下的方法：

```
public void DeleteCar(int id)
{
  // 获取要删除的汽车的ID，然后删除
  string sql = string.Format("Delete from Inventory where CarID = '{0}'",
    id);
  using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    try
    {
      cmd.ExecuteNonQuery();
    }
    catch(SqlException ex)
    {
      Exception error = new Exception("Sorry! That car is on order!", ex);
      throw error;
    }
  }
}
```

21.10.4 增加更新逻辑

谈到Inventory表中的既有记录，第一个问题就是我们究竟希望调用者改变什么数据？汽车的颜色？昵称还是制造商？或者所有？当然，一个让调用者充分灵活的方式就是只定义一个方法，接受表示任何SQL语句的string类型，但是这太冒险了。

理想情况下，我们可能会设置一组方法，允许调用者以各种方式来更新记录。然而，为了做一个简单的数据访问库，我们会定义一个方法，允许调用者更新某辆汽车的昵称，如下所示：

```
public void UpdateCarPetName(int id, string newPetName)
{
  // 获取要修改的汽车的ID以及新的昵称
  string sql = string.Format("Update Inventory Set PetName = '{0}' Where CarID = '{1}'",
    newPetName, id);

  using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    cmd.ExecuteNonQuery();
  }
}
```


21.10.5 增加选择逻辑

下一个要增加的是选择方法。在本章之前我们看到，一个数据提供程序的数据读取器对象可以使用只能向前读的服务器端游标来选取记录。调用Read()方法后，我们就可以以统一的方式处理每一个记录。这没什么问题，但是我们需要处理如何将这些记录返回给应用程序调用层这个问题。

一个方法就是用从Read()方法获取的数据填充多维数组（或诸如泛型List<NewCar>的此类对象）并返回。另外一种方法（当前示例会采用）就是从Inventory表获取数据：

```
public List<NewCar> GetAllInventoryAsList()
{
  // 这将保存记录
  List<NewCar> inv = new List<NewCar>();

  // 准备命令对象
  string sql = "Select * From Inventory";
  using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    SqlDataReader dr = cmd.ExecuteReader();
    while (dr.Read())
    {
      inv.Add(new NewCar
      {
        CarID = (int)dr["CarID"],
        Color = (string)dr["Color"],
        Make = (string)dr["Make"],
        PetName = (string)dr["PetName"]
      });
    }
    dr.Close();
  }
  return inv;
}
```

还有另一种返回System.Data.DataTable对象的方法，它实际上是ADO.NET非连接层的一部分。我们将在下一章中全面介绍非连接层。而现在，你只需要理解DataTable是一个表示表格式数据（如数据表格中的网格）的类类型。

DataTable类表示行和列的数据集合。虽然这些集合可以以编程方式进行填充，但DataTable类型还提供了一个Load()方法，它会使用数据读取器对象自动填充这些集合。考虑下面的方法，将Inventory中的数据以DataTable的形式返回：

```
public DataTable GetAllInventoryAsDataTable()
{
```

```
        // 它会保存记录
        DataTable inv = new DataTable();

        // 准备命令对象
        string sql = "Select * From Inventory";
        using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
        {
          SqlDataReader dr = cmd.ExecuteReader();
          // 用读取器的数据填充DataTable并且清理
          inv.Load(dr);
          dr.Close();
        }
        return inv;
      }
```

21.10.6　使用参数化的命令对象

现在InventoryDAL类型的插入、更新和删除逻辑的每一个SQL查询都是用字符串字面量硬编码的。读者可能知道，参数化查询能像对象那样处理SQL参数，而不是像一堆文本一样。而且，参数化查询执行起来比纯文本的SQL语句快多了，它只需要解析一次（而不是像SQL语句那样每次被分配到CommandText属性都要解析）。同样，参数化查询也能消除SQL注入攻击（非常著名的数据访问安全问题）的隐患。

为支持参数化查询，ADO.NET命令对象使用一个集合来保存参数对象。这个集合默认是空的，可以添加任意多的参数对象并映射到SQL语句中的占位符参数。如果需要把SQL查询中的参数和命令对象参数集合中某一个成员关联的话，只需要在SQL文本参数前面加@符号（微软SQL Server是这样的，不是所有的DBMS都支持@符号）。

使用DbParameter类型指定参数

在建立参数化查询以前，先来看一下DbParameter类型（提供程序参数对象的基类）。这个类有许多属性让你来配置参数的名字、大小、类型和其他一些诸如参数传递方向等的特性。表21-7列举了DbParameter类型的一些主要属性。

表21-7　DbParameter类型的主要成员

属　性	作　用
DbType	从数据源获取或设置原始数据类型，以CLR数据类型呈现
Direction	获取或设置一个值，该值指示参数是只可输入、只可输出、双向还是返回值参数
IsNullable	获取或设置一个值，该值指示参数是否接受空值
ParameterName	获取或设置DbParameter的名称
Size	获取或设置列中数据的最大尺寸（只对文本数据有用）
Value	获取或设置该参数的值

为了演示如何填充与DBParameter兼容的对象的命令对象集合，让我们重新修改前面InsertAuto()方法来使用参数（也可以对其余的SQL方法做类似的修改，但在本例中没必要这样做）：

```
public void InsertAuto(int id, string color, string make, string petName)
{
  // 注意SQL查询中的“占位符”
  string sql = string.Format("Insert Into Inventory" +
      "(CarID, Make, Color, PetName) Values" +
      "(@CarID, @Make, @Color, @PetName)");

  // 这个命令有内部参数
  using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    // 填充参数集合
    SqlParameter param = new SqlParameter();
    param.ParameterName = "@CarID";
    param.Value = id;
    param.SqlDbType = SqlDbType.Int;
    cmd.Parameters.Add(param);

    param = new SqlParameter();
    param.ParameterName = "@Make";
    param.Value = make;
    param.SqlDbType = SqlDbType.Char;
    param.Size = 10;
    cmd.Parameters.Add(param);

    param = new SqlParameter();
    param.ParameterName = "@Color";
    param.Value = color;
    param.SqlDbType = SqlDbType.Char;
    param.Size = 10;
    cmd.Parameters.Add(param);

    param = new SqlParameter();
    param.ParameterName = "@PetName";
    param.Value = petName;
    param.SqlDbType = SqlDbType.Char;
    param.Size = 10;
    cmd.Parameters.Add(param);

    cmd.ExecuteNonQuery();
  }
}
```

同样，注意SQL查询包含4个嵌入的占位符，每一个都以@标识为前缀。使用SqlParameter类型，我们就可以使用ParameterName属性以强类型格式来映射每一个占位符并指定各种细节（值、数据类型和大小等）。参数对象都准备好之后，就可以通过调用Add()方法加到命令对象的集合中。

说明 在这里，我使用了各种属性来创建参数对象。但要知道，这个参数对象支持许多重载构造函数，允许我们设置各种属性的值（这样能产生更简洁的代码）。还要知道，Visual Studio提供了许多图形设计器，它们可以根据需要生成很多无技术含量的参数相关的代码（参见第22章）。

虽然构建参数化查询总是需要很多代码，但是我们可以更方便地以编程方式调试SQL语句，并且性能更好。虽然我们完全可以在调用SQL查询时使用这项技术，但是参数化查询在我们希望调用存储过程时更有用。

21.10.7 执行存储过程

存储过程是存储在数据库内的一段已命名的SQL代码，你可以构造存储过程，返回一组行或标量数据类型，或进行其他有意义的处理（如插入、更新、删除），也可以接受一些可选参数。其实它就像是一个存储在数据库内的功能模块，和那些二进制数据对象有明显区别。AutoLot数据库定义了一个存储过程GetPetname，按如下方式对它进行格式化：

```
GetPetName
@carID int,
@petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarID = @carID
```

下面是InventoryDAL类型的最后一个方法，它调用了GetPetname存储过程：

```
public string LookUpPetName(int carID)
{
  string carPetName = string.Empty;

  // 设定存储过程名
  using (SqlCommand cmd = new SqlCommand("GetPetName", this.sqlCn))
  {
    cmd.CommandType = CommandType.StoredProcedure;

    // 输入参数
    SqlParameter param = new SqlParameter();
    param.ParameterName = "@carID";
    param.SqlDbType = SqlDbType.Int;
    param.Value = carID;

    // 默认的方向即为Input，但为了更清楚
    param.Direction = ParameterDirection.Input;
    cmd.Parameters.Add(param);

    // 输出参数
    param = new SqlParameter();
    param.ParameterName = "@petName";
    param.SqlDbType = SqlDbType.Char;
    param.Size = 10;
    param.Direction = ParameterDirection.Output;
    cmd.Parameters.Add(param);

    // 执行存储过程
    cmd.ExecuteNonQuery();

    // 返回输出参数
    carPetName = (string)cmd.Parameters["@petName"].Value;
  }
  return carPetName;
}
```

调用存储过程的一个重要方面就是命令对象表示SQL语句（默认）或者存储过程名称。如果希望通知命令对象调用一个存储过程，我们就传入过程名称（作为构造函数实参或通过CommandText属性），并且必须设置CommandType属性为值CommandType.StoredProcedure（如果不这样做，我们就会收到一个运行时异常，因为命令对象在默认情况下接受SQL语句）：

```
SqlCommand cmd = new SqlCommand("GetPetName", this.sqlCn);
cmd.CommandType = CommandType.StoredProcedure;
```

接着要注意，参数对象的Direction属性可以为每一个传入存储过程的参数指定传递方向（如输入参数、输出参数、in／out参数或返回值）。像以前那样，把每一个参数对象加入命令对象的参数集合中：

```
// 输入参数
SqlParameter param = new SqlParameter();
param.ParameterName = "@carID";
param.SqlDbType = SqlDbType.Int;
param.Value = carID;
param.Direction = ParameterDirection.Input;
cmd.Parameters.Add(param);
```

最后，通过调用ExecuteNonQuery()完成存储过程之后，可以通过调查命令对象的参数集合并且做相应的强制类型转换来获取输出参数的值。

```
// 返回输出参数
carPetName = (string)cmd.Parameters["@petName"].Value;
```

至此，AutoLotDAL.dll数据访问类库的首次迭代完成了。使用这个程序集，我们可以构建任何类型的前端（基于控制台、桌面GUI或者基于HTML的Web应用程序）来显示和编辑数据。由于还没有研究如何构建GUI，我们会从一个新的控制台应用程序来测试我们的类库。

源代码 AutoLotDAL项目的源代码位于Chapter 21子目录下。

21.11 创建控制台 UI 前端

新建一个控制台应用程序AutoLotCUIclient。在创建了新的项目之后，请确保添加AutoLotDAL.dll程序集和System.Configuration.dll的引用，并且添加如下using语句到C#代码文件中：

```
using AutoLotConnectedLayer;
using System.Configuration;
using System.Data;
```

接着，插入一个新的App.config文件到项目中，它包含一个用于连接到AutoLot数据库实例的<connectionString>元素，例如：

```
<configuration>
  <connectionStrings>
    <add name ="AutoLotSqlProvider" connectionString =
    "Data Source=(local)\SQLEXPRESS;
     Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
</configuration>
```

21.11.1 实现Main()方法

Main()方法负责响应用户的一系列行为，并且通过switch语句执行请求。这个程序允许用户输入如下命令。

- I：插入新的记录到Inventory表。
- U：更新Inventory表的既有记录。
- D：从Inventory表中删除既有记录。
- L：使用数据读取器显示当前库存。
- S：为用户显示这些选项。
- P：根据汽车ID查询昵称。
- Q：退出程序。

每一个可能的选项都由Program类中对应的唯一一个静态方法处理。下面是Main()方法的完整实现。注意，每一个从do/while循环中调用的方法（除了ShowInstruction()方法之外）都接受InventoryDAL对象作为唯一参数：

```
static void Main(string[] args)
{
  Console.WriteLine("***** The AutoLot Console UI *****\n");

  // 从App.config获取连接字符串
  string cnStr =
    ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].ConnectionString;
  bool userDone = false;
  string userCommand = "";

  // 创建InventoryDAL对象
  InventoryDAL invDAL = new InventoryDAL();
  invDAL.OpenConnection(cnStr);

  // 不断请求输入，直到用户按下Q键
  try
  {
    ShowInstructions();
    do
    {
      Console.Write("\nPlease enter your command: ");
      userCommand = Console.ReadLine();
      Console.WriteLine();
      switch (userCommand.ToUpper())
      {
      case "I":
        InsertNewCar(invDAL);
        break;
      case "U":
        UpdateCarPetName(invDAL);
        break;
      case "D":
        DeleteCar(invDAL);
        break;
      case "L":
        ListInventory(invDAL);
        break;
      case "S":
        ShowInstructions();
        break;
      case "P":
        LookUpPetName(invDAL);
        break;
```

```
        case "Q":
          userDone = true;
          break;
        default:
          Console.WriteLine("Bad data!  Try again");
          break;
      }
    } while (!userDone);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
  finally
  {
    invDAL.CloseConnection();
  }
}
```


21.11.2　实现ShowInstructions()方法

ShowInstructions()方法做的事情如下所示：

```
private static void ShowInstructions()
{
  Console.WriteLine("I: Inserts a new car.");
  Console.WriteLine("U: Updates an existing car.");
  Console.WriteLine("D: Deletes an existing car.");
  Console.WriteLine("L: Lists current inventory.");
  Console.WriteLine("S: Shows these instructions.");
  Console.WriteLine("P: Looks up pet name.");
  Console.WriteLine("Q: Quits program.");
}
```

21.11.3　实现ListInventory()方法

根据构建数据访问库的方式，可以通过两种方式中的任何一种实现ListInventory()方法。你应该记得，InventoryDAL对象的GetAllInventoryAsDataTable()方法返回DataTable对象。你可以像下面这样实现ListInventory()方法：

```
private static void ListInventory(InventoryDAL invDAL)
{
  // 获取库存列表
  DataTable dt = invDAL.GetAllInventoryAsDataTable();

  // 将DataTable传递给辅助函数用于显示数据
  DisplayTable(dt);
}
```

DisplayTable()辅助方法使用传入DataTable的Rows和Columns属性显示表数据（DataTable对象的完整细节会在下一章中介绍，在这里先不要纠缠于细节）：

```
private static void DisplayTable(DataTable dt)
{
  // 输出列名
  for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
```

```
    {
      Console.Write(dt.Columns[curCol].ColumnName + "\t");
    }
    Console.WriteLine("\n----------------------------------");

    // 输出DataTable
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
      for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
      }
      Console.WriteLine();
    }
  }
```

如果希望调用InventoryDAL的GetAllInventoryAsList()方法，可以实现ListInventoryViaList()方法，如下所示：

```
private static void ListInventoryViaList(InventoryDAL invDAL)
{
  // 获取库存列表
  List<NewCar> record = invDAL.GetAllInventoryAsList();

  foreach (NewCar c in record)
  {
    Console.WriteLine("CarID: {0}, Make: {1}, Color: {2}, PetName: {3}",
      c.CarID, c.Make, c.Color, c.PetName);
  }
}
```

21.11.4 实现DeleteCar()方法

删除既有汽车很简单，只需要向用户询问要删除的汽车ID，然后把它传给InventoryDAL类型的DeleteCar()方法，如下所示：

```
private static void DeleteCar(InventoryDAL invDAL)
{
  // 获取要删除的汽车ID
  Console.Write("Enter ID of Car to delete: ");
  int id = int.Parse(Console.ReadLine());

  // 以防违反引用完整性
  try
  {
    invDAL.DeleteCar(id);
  }
  catch(Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

21.11.5 实现InsertNewCar()方法

插入新记录到Inventory表也很简单，只需要向用户请求新数据（通过调用Console.ReadLine()），

并把数据传入InventoryDAL的InsertAuto()方法，如下所示：

```
private static void InsertNewCar(InventoryDAL invDAL)
{
  // 首先获取用户数据
  int newCarID;
  string newCarColor, newCarMake, newCarPetName;

  Console.Write("Enter Car ID: ");
  newCarID = int.Parse(Console.ReadLine());

  Console.Write("Enter Car Color: ");
  newCarColor = Console.ReadLine();

  Console.Write("Enter Car Make: ");
  newCarMake = Console.ReadLine();

  Console.Write("Enter Pet Name: ");
  newCarPetName = Console.ReadLine();

  // 现在传入数据访问类库
  invDAL.InsertAuto(newCarID, newCarColor, newCarMake, newCarPetName);
}
```

我们还重载了InsertAuto()方法，它接收NewCar对象，而不是一些独立的参数。因此，你可以这样实现InsertNewCar()方法：

```
private static void InsertNewCar(InventoryDAL invDAL)
{
  // 首先获取用户数据
...

  // 现在传入数据访问库
  NewCar c = new NewCar { CarID = newCarID, Color = newCarColor,
                          Make = newCarMake, PetName = newCarPetName };
  invDAL.InsertAuto(c);
}
```

21.11.6 实现UpdateCarPetName()方法

下面的UpdateCarPetName()的实现也非常相似：

```
private static void UpdateCarPetName(InventoryDAL invDAL)
{
  // 首先获取用户数据
  int carID;
  string newCarPetName;

  Console.Write("Enter Car ID: ");
  carID = int.Parse(Console.ReadLine());
  Console.Write("Enter New Pet Name: ");
  newCarPetName = Console.ReadLine();

  // 现在传入数据访问类库
  invDAL.UpdateCarPetName(carID, newCarPetName);
}
```

21.11.7　实现LookUpPetName()

获取给定汽车昵称的方法与前面的方法类似，因为数据访问库封装了所有低级别的ADO.NET调用：

```
private static void LookUpPetName(InventoryDAL invDAL)
{
  // 获取要查找的汽车的ID
  Console.Write("Enter ID of Car to look up: ");
  int id = int.Parse(Console.ReadLine());
  Console.WriteLine("Petname of {0} is {1}.",
    id, invDAL.LookUpPetName(id).TrimEnd());
}
```

有了这个方法，基于控制台的前端就基本完成了。现在该运行程序测试各个方法了。以下是测试L、P、Q命令的部分输出结果：

```
***** The AutoLot Console UI *****

I: Inserts a new car.
U: Updates an existing car.
D: Deletes an existing car.
L: Lists current inventory.
S: Shows these instructions.
P: Looks up pet name.
Q: Quits program.

Please enter your command: L

CarID   Make    Color   PetName
----------------------------------
83      Ford    Rust    Rusty
107     Ford    Red     Snake
678     Yugo    Green   Clunker
904     VW      Black   Hank
1000    BMW     Black   Bimmer
1001    BMW     Tan     Daisy
1992    Saab    Pink    Pinkey

Please enter your command: P

Enter ID of Car to look up: 904
Petname of 904 is Hank.

Please enter your command: Q

Press any key to continue . . .
```

源代码　AutoLotCUIClient项目的源代码位于Chapter 21子目录下。

21.12 数据库事务

在结束对ADO.NET连接层的研究前，我们来看一下数据库事务的概念。简单来说，事务是一组数据库操作，作为一个整体，它们要么全部完成，要么全部失败。我们可以想象，事务对于确保表数据的安全、有效以及一致性来说非常重要。

如果数据库操作包含与多个表或多个存储过程（或数据库原子的组合）的交互，事务就非常重要。经典的事务示例包括在两个银行账户之间转账的过程。例如，如果我们从储蓄账户转账500美元到支票账户，如下步骤应该以事务方式发生：

- 银行应该从储蓄账户移除500美元；
- 银行应该为支票账户增加500美元。

如果钱从储蓄账户移走了但又没有转移到支票账户，这确实是一件糟糕的事情（由于银行方面的某些错误），因为我们现在损失了500美元！然而，如果这些步骤放入数据库事务，DBMS就会确保所有相关步骤以一个整体发生。如果任何一个事务失败，整个操作就会回滚到原始状态。另一方面，如果所有步骤都成功，事务就会被提交。

说明 你可能在研究事务相关资料时已经听说过ACID这个缩写。这表示一个正规事务的关键属性，具体而言就是原子性（全有或全无）、一致性（数据在整个事务中保持稳定）、隔离性（事务之间不会互相影响）和持久性（事务会被保存和记录日志）。

其实，.NET平台以各种形式支持事务。对本章来说，最重要的就是ADO.NET数据提供程序的事务对象（在这里就是`System.Data.SqlClient`中的`SqlTransaction`）。此外，.NET基础类库提供了许多API来支持事务。

- `System.EnterpriseServices`：这个命名空间（位于System.EnterpriseServices.dll程序集）提供的类型允许我们和COM+运行库层结合使用，包括对分布式事务的支持。
- `System.Transactions`：这个命名空间（位于System.Transactions.dll程序集）包含的类允许我们为各种服务写事务性应用程序和资源管理器（如MSMQ、ADO.NET、COM+等）。
- WCF（Windows Communication Foundation）：WCF API提供的服务使事务便于处理各种分布式绑定类。
- WF（Windows Workflow Foundations）：WF API为工作流活动提供事务性支持。

除了.NET基础类库中现成的事务支持外，还可以使用数据管理系统中的SQL语言本身来实现事务。例如，我们可以编写存储过程来利用`BEGIN TRANSACTION`、`ROLLBACK`和`COMMIT`语句。

21.12.1 ADO.NET事务对象的主要成员

虽然事务相关类型在基础类库中有很多，但是我们将着重研究ADO.NET数据提供程序中的事务对象，它们都从`DBTransaction`派生并且实现了`IDbTransaction`接口。回忆一下本章开头部分，`IDbTransaction`定义了一批成员，如下所示：

```
public interface IDbTransaction : IDisposable
{
  IDbConnection Connection { get; }
  IsolationLevel IsolationLevel { get; }

  void Commit();
  void Rollback();
}
```

注意Connection属性，它会返回对初始化当前事务的连接对象的引用（可以看到，我们从某个连接对象获得事务对象）。Commit()方法在所有数据库操作都完成后才调用，为此，所有待定的改变都会持久化到数据库中。相反，Rollback()方法会在出现运行时异常时进行调用，它会通知DMBS来放弃所有待定的改变，保留原始数据。

说明　事务对象的IsolationLevel属性允许我们指定事务在遇到其他并行事务时如何处理。默认情况下，事务会被完整隔离直到提交。更多有关IsolationLevel枚举值的细节，请参考.NET Framework 4.5 SDK文档。

除了IDbTransaction接口定义的成员，SqlTransaction类型还定义了一个叫做Save()的成员，它允许我们定义保存点。这个概念允许我们把失败的事务回滚到一个保存点，而不是回滚整个事务。其实，在调用SqlTransaction对象的Save()方法时，我们可以指定一个友好的字符串名。在调用Rollback()时，我们就可以指定相同的名字来有效执行部分回滚。如果调用的Rollback()不带参数，所有的待定改变会被回滚。

21.12.2　为AutoLot数据库添加CreditRisks表

为了说明ADO.NET事务的使用，首先使用Visual Studio的服务器资源管理器向AutoLot数据库添加一个叫CreditRisks的新表，其中的列和本章前面创建的Customers表完全一样（作为主键的CustID、FirstName和LastName）。顾名思义，CreditRisks用于记录没有通过信用检查的客户（如图21-15所示）。

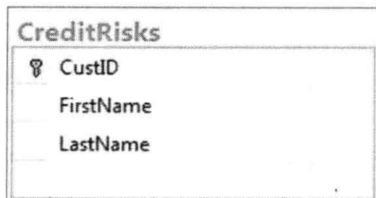


图21-15　CreditRisks表

和之前描述的从储蓄账户转账到支票账户的示例相似，把风险客户从Customers表转移到CreditRisks表也应该在事务范围中发生（毕竟，我们会希望记住那些不诚信的客户的ID和名字）。具体而言，我们需要确保这两个数据库操作要么都发生，要么都不发生：从当前Customers表删除当前信用风险和把它们加入到CreditRisks表。

说明 在生产环境下，没有必要构建完整的数据库表来获取高风险客户。你可以在已知的Customers表中添加Boolean列IsCreditRisk。但这个新表可以执行简单的事务。

21.12.3 为InventoryDAL添加事物方法

为了演示如何以编程方式使用ADO.NET事务，打开我们在本章前面创建的AutoLotDAL代码类库项目。新增一个公共方法ProcessCreditRisk()到InventoryDAL类中，它会如下处理接收到的信用风险（注意本例中不使用参数化查询未简化实现，但是你会想为生产级方法使用这种查询）:

21

```
// InventoryDAL类的新成员
public void ProcessCreditRisk(bool throwEx, int custID)
{
  // 首先，基于客户ID查询当前的名字
  string fName = string.Empty;
  string lName = string.Empty;
  SqlCommand cmdSelect = new SqlCommand(
    string.Format("Select * from Customers where CustID = {0}", custID), sqlCn);
  using (SqlDataReader dr = cmdSelect.ExecuteReader())
  {
    if(dr.HasRows)
    {
      dr.Read();
      fName = (string)dr["FirstName"];
      lName = (string)dr["LastName"];
    }
    else
      return;
  }

  // 创建表示每一步操作的命令对象
  SqlCommand cmdRemove = new SqlCommand(
    string.Format("Delete from Customers where CustID = {0}", custID), sqlCn);

  SqlCommand cmdInsert = new SqlCommand(string.Format("Insert Into CreditRisks" +
                  "(CustID, FirstName, LastName) Values" +
                  "({0}, '{1}', '{2}')", custID, fName, lName), sqlCn);

  // 我们会从连接对象获得
  SqlTransaction tx = null;
  try
  {
    tx = sqlCn.BeginTransaction();

    // 将命令加入到事务
    cmdInsert.Transaction = tx;
    cmdRemove.Transaction = tx;

    // 执行命令
    cmdInsert.ExecuteNonQuery();
    cmdRemove.ExecuteNonQuery();

    // 模拟错误
    if (throwEx)
    {
```

```
      throw new Exception("Sorry!  Database error! Tx failed...");
    }

    // 提交
    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    // 有任何错误都会回滚事务
    tx.Rollback();
  }
}
```

在这里，我们使用传入的bool参数来表示是否在处理不诚信客户时会抛出一个异常。这能让我们轻松模拟会导致数据库事务失败的不可预料的情况。很明显，这只是出于演示目的，一个真正的数据库事务方法当然不会允许调用者来强制逻辑失败！

注意，我们使用两个sqlcommand对象来表示事务中的每一步。根据传入的custID参数获取了客户的姓和名之后，通过BeginTransaction()方法从连接对象获取有效的SqlTransaction对象。接下来，最重要的是，我们必须通过把Transaction属性指定给刚才获得的事务对象来登记每个命令对象。如果不这样做，插入/删除逻辑就不会处于事务上下文之中。

调用了每一个命令的ExecuteNonQuery()之后，如果（仅当）bool参数的值是true，我们就抛出一个异常。在这种情况下，所有待定的数据库操作都会回滚。如果不抛出异常，所有的步骤都会在我们调用Commit()时被提交到数据库表中。编译修改后的AutoLotDAL项目以确保没有任何拼写错误。

21.12.4　测试数据库事务

虽然现在我们就可以更新之前的AutoLotCUIClient应用程序，新增一个调用ProcessCreditRisk()方法的选项，但我们还是新建一个控制台应用程序AdoNetTransaction。添加AutoLotDAL.dll程序集的引用，并且导入AutoLotConnectedLayer命名空间。

然后，右击ServerExplorer中的表图标，选择Show Table Data，打开Customers表来输入数据。新增一个低信用的客户，例如，

- CustID: 333
- FirstName: Homer
- LastName: Simpson

最后，按如下所示更新我们的Main()方法：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Transaction Example *****\n");

  // 让事务成功或失败的简单方式
  bool throwEx = true;
  string userAnswer = string.Empty;

  Console.Write("Do you want to throw an exception (Y or N): ");
  userAnswer = Console.ReadLine();
  if (userAnswer.ToLower() == "n")
  {
```

```
      throwEx = false;
    }

    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;Integrated Security=SSPI;" +
      "Initial Catalog=AutoLot");

    // 处理客户333
    dal.ProcessCreditRisk(throwEx, 333);

    Console.WriteLine("Check CreditRisk table for results");
    Console.ReadLine();
  }
```

如果运行程序并选择抛出一个异常，就会发现Homer没有从Customers表移除，因为整个事务被回滚了。然而，如果不抛出异常，就会发现Customer ID 333不再存在于Customers表中，而是被放到了CreditRisks表中。

源代码 AdoNetTransaction项目的源代码位于Chapter 21子目录下。

21.13 小结

ADO.NET是.NET平台的原生数据访问技术，它可以以三种不同的方式进行使用：连接式、断开连接式或通过Entity框架。在本章中，我们研究了连接层并进而理解了数据提供程序的作用，它们本质上是几个抽象基类（在System.Data.Common命名空间中）和接口类型（在System.Data命名空间中）的具体实现。我们已经看到，可以使用ADO.NET数据提供程序工厂模型来构建提供程序无关的代码。

通过使用连接对象、事务对象、命令对象以及连接层的数据读取器对象，我们就可以选择、更新、插入和删除记录。另外你应该记得，命令对象支持内部参数集合，它可以为SQL查询增加类型安全，并且对于触发存储过程也非常有用。

第22章 ADO.NET之二：断开连接层

第21章介绍了ADO.NET的连接层，它允许使用连接、命令和数据提供程序的数据读取器对象提交SQL语句到数据库。在本章中，我们会介绍ADO.NET的断开连接层。如果我们使用它，就可以使用System.Data命名空间的许多成员（主要是DataSet、DataTable、DataRow、DataColumn、DataView和DataRelation）在调用层建模内存中的数据库数据。这样就使我们误以为调用层持续连接着外部数据源，而其实只是在操作关系数据的本地副本。

虽然完全可以使用ADO.NET的断开功能而不连接任何关系数据库，但是我们通常会使用数据提供程序的数据适配器对象来填充DataSet对象。我们会看到，数据适配器对象就像客户层和关系数据库之前的桥梁。使用这些对象，我们就可以获取DataSet对象，操作它们的内容，把修改后的记录发送回去处理。这样就能产生高度可扩展的数据相关的.NET应用程序。

本章还将使用Windows Forms GUI桌面应用程序演示一些数据绑定技术，并介绍强类型DataSet的作用。我们会更新第21章中创建的AutoLotDAL.dll数据类库，使用ADO.NET断开连接层的新命名空间。最后，我们会研究LINQ to DataSet的作用，它允许将LINQ查询应用于内存数据缓存。

说明 你将在本书后面学习WPF和ASP.NET应用程序的不同的数据绑定技术。

22.1 ADO.NET 断开连接层

通过前面的章节我们知道，连接层可以通过连接、命令、数据读取器对象来和数据库交互。仅仅使用这几个简单的类型你就能“尽情享用”选择、插入、更新、删除记录的操作［也包括调用存储过程或执行其他数据操作（使用DDL创建表，使用DCL赋予权限）］。实际上ADO.NET的内容你才看了一半，前面曾说过，ADO.NET对象模型还包括断开式模式。

使用断开连接层，就可以通过内存中的对象模型来构建关系数据。除了构建表格形式的行和列，System.Data中的类型允许我们表示表关系、列约束、主键、视图以及其他数据库基元。此外，一旦我们构建了数据，就可以应用过滤器，提交内存中的查询以及以XML或二进制格式持久化（或加载）数据。我们甚至可以在不与DBMS连接的情况下做所有这些事情（因此也就称为断开连接层），这可以通过从本地XML文件加载数据或在代码中手动构建DataSet来实现。

说明 第23章将学习ADO.NET Entity Framework，它是基于本章介绍的断开连接层的概念构建的。

当使用ADO.NET的断开式访问方式的时候,不需要连接到数据库,但仍然会使用连接和命令对象。另外，我们还会补充一个叫做数据适配器的特殊对象（扩展自DbDataAdapter抽象类）来获取和更新数据。与连接层不同，这里的数据通过数据适配器而不是数据读取器获取，并且数据适配器对象通过DataSet对象（或者更确切地说，是DataSet中的DataTable对象）在调用者和数据源之间传递数据。DataSet可以包含任意多的DataTable对象，而DataTable对象又是DataRow和DataColumn对象的集合。

数据提供程序的数据适配器对象自动处理数据库连接。为了增加性能，数据适配器会尽可能缩短连接打开的总时间。一旦调用者获得了DataSet对象，它会立即关闭DBMS的连接，仅在本地留下一个远程数据的本地副本。调用者可以任意对DataTable进行插入、修改和删除操作，但物理数据库不会被更新，直到调用者显式将DataSet中的DataTable提交到到数据适配器才进行更新。简而言之，DataSet造成了客户端总是连接着数据库的假相,其实所有的操作都是对一个内存中的数据库进行的(如图22-1所示)。

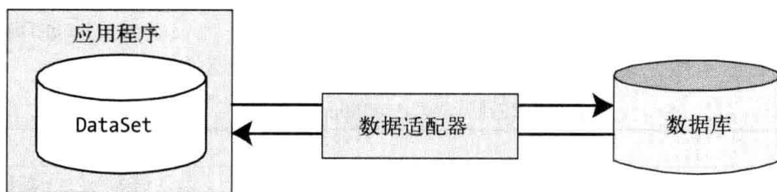


图22-1 数据适配器对象从客户端提交DataSet、返回DataSet到客户端

既然断开连接层的亮点在于DataSet，下面我们的任务就是去研究如何手动操作DataSet。掌握了这些，对你来说操作从数据适配器对象返回的DataSet的数据就轻而易举了。

22.2 DataSet 的作用

前面提到过，DataSet是对关系数据的一种在内存中的表现形式。说得具体一点，其实DataSet类型内部包含了3个强类型的集合（如图22-2所示）。

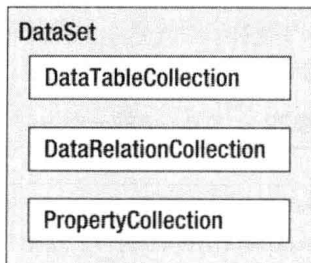


图22-2 剖析DataSet

DataSet的Tables属性允许访问包含独立DataTable的DataTableCollection。DataSet所使用的另外一个重要的集合是DataRelationCollection，既然DataSet是数据库的一个断开连接的副本，DataRelationCollection可以通过编程来表示各表之间的父子关系。比如，使用DataRelation类型对两

个外键约束关联的表建立联系。然后能通过Relations属性把这个对象加到DataRelationCollection。这样，你就能非常方便地在这些关联表中搜索数据。在本章后续部分，会看到详细的实现。

ExtendedProperties属性提供了PropertyCollection对象的访问，通过它能把额外的名称/值信息关联到DataSet，当然这些信息可以是任何东西，即使它们和数据本身没有任何关系。比如说你能把你公司的名字关联到DataSet，表现为一种在内存中的元数据。其他的一些扩展属性的用途还包括时间戳、加密的密码（必须提供密码才能访问DataSet内容）和数据刷新时间的数值等。

说明　DataTable和DataColumn类同样通过ExtendedProperties属性支持扩展属性。

22.2.1 DataSet 的主要属性

在探究其他的一些编程细节以前，先来看一下DataSet的一些核心成员。除了Tables、Relations和ExtendedProperties属性，表22-1还列举了其他的一些有意思的属性。

表22-1　DataSet的一些属性

属　　性	作　　用
CaseSensitive	指示DataTable对象中的字符串比较是否区分大小写。默认值为false（默认情况下不区分大小写）
DataSetName	表示DataSet的一个友好名，通常通过构造参数指定
EnforceConstraints	获取或设置一个值，该值指示在尝试执行任何更新操作时是否遵循约束规则
HasErrors	获取一个值，指示在此DataSet中任何DataTable对象中的数据行是否存在错误
RemotingFormat	定义DataSet在传输时内容的序列化方式［二进制序列化或XML序列化（默认值）］

22.2.2 DataSet 的主要方法

DataSet的方法实现某些前面属性提到的那些功能，除了和XML数据流交互以外，DataSet提供了一些方法让你复制DataSet的内容，在内部表之间导航，另外包括一些诸如设置批量更新起始/结束点的功能。表22-2描述了其中的一些核心方法。

表22-2　DataSet的一些方法

方　　法	作　　用
AcceptChanges()	提交自加载此DataSet或上次调用AcceptChanges()以来对其进行的所有更改
Clear()	通过移除所有DataTable表中的所有行来清除任何DataSet的数据
Clone()	复制DataSet的结构，而不是DataSet的数据，包括所有DataTable架构、关系和约束
Copy()	复制该DataSet的结构和数据
GetChanges()	获取DataSet的副本，该副本包含自上次加载以来或自调用AcceptChanges()以来对该数据集进行的所有更改。重载该方法以便只得到执行、已修改的行或已删除的行
HasChanges()	获取一个值，该值指示DataSet是否有更改，包括新增行、已删除的行或已修改的行

（续）

方　法	作　用
Merge()	将指定的DataSet合并到当前的DataSet中
ReadXml()	基于XML架构和从流中读取的数据，定义DataSet对象的结构并用数据填充它
RejectChanges()	回滚自创建DataSet以来或上次调用AcceptChanges()以来对其进行的所有更改
WriteXml()	把DataSet的内容写入一个有效的数据流

22.2.3 构建DataSet

现在对DataSet的作用了解得更清楚了，下面建立一个新的控制台程序SimpleDataSet并导入System.Data命名空间。在Main()方法里定义一个新的DataSet对象，它包含3个扩展属性，分别代表公司名、标识符（表示为System.Guid类型）和时间戳，如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with DataSets *****\n");

  // 建立DataSet对象并添加一些属性
  DataSet carsInventoryDS = new DataSet("Car Inventory");

  carsInventoryDS.ExtendedProperties["TimeStamp"] = DateTime.Now;
  carsInventoryDS.ExtendedProperties["DataSetID"] = Guid.NewGuid();
  carsInventoryDS.ExtendedProperties["Company"] =
    "Mikko's Hot Tub Super Store";
  Console.ReadLine();
}
```

说明　GUID（全局唯一标识符）是静态唯一的128位数字。

在任何情况下，只有你在DataSet中插入几个DataTable时，DataSet对象才会有意思。因此接下来就要研究DataTable的内部结构，我们先从DataColumn类型开始。

22.3 使用 DataColumn

DataColumn类型表示DataTable中的一个单列。一般来说，一组DataColumn类型绑定到一个特定的DataTable，组合成一个表的基本结构信息。比如你想为AutoLot数据库建立一个Inventory表（见第21章），就要建立4个DataColumn，一个DataColumn对应一列（CarID、Make、Color和PetName）。在建立完DataColumn对象以后，需要把这些DataColumn放到DataTable类型的列集合中（通过DataTable的Columns属性）。

如果了解关系数据库的话，你可能知道数据库表中的某列能被指定一组约束（比如配置成主键，设置默认值，配置成只读等）。还有，数据表中的每列必须映射到一个基层的数据类型，比如说Inventory表的结构需要映射CarID列为数字型，映射Make、Color、PetName为字符数组型。DataColumn类有许多允许配置这些东西的属性。表22-3简要介绍了一些重要属性。

表22-3　DataColumn的属性

属　　性	作　　用
AllowDBNull	该属性用来指示对于表中的行，此列中是否允许空值，默认为true
AutoIncrement、AutoIncrementSeed和AutoIncrementStep	这些属性用来配置该列的自增行为，当需要确保该列值的唯一性（比如说主键）时将会非常有用。默认情况下，DataColumn不支持自增
Caption	该属性用来获取或设置要显示的列的标题，它允许我们为数据库列名定义一个用户友好的版本
ColumnMapping	该属性决定当使用DataSet.WriteXml()来把DataSet保存到XML文件时，该列将以什么样的形式呈现。可以把数据列写成XML元素、XML特性或文本的形式
ColumnName	该属性用来获取或设置列集合中的列名（在DataTable内部以什么样的名字呈现）。如果不显式设置ColumnName，默认的列名格式是"Column"字符和（n+1）后缀的形式（比如Column1、Column2、Column3等）
DataType	该属性定义了存储在列中的数据类型（Boolean、string、float等）
DefaultValue	该属性用来获取或设置在创建新行时列的默认值
Expression	该属性用来获取或设置表达式，用于筛选行、计算列中的值或创建聚合列
Ordinal	该属性用来以数字形式获取DataTable中列在Columns集合中的位置
ReadOnly	该属性指示如果向表中添加了行，列是否还允许更改，默认为false
Table	该属性用来获取DataColumn所属的DataTable
Unique	该属性用来获取或设置一个值，指示列的每一行中的值是否必须是唯一/可重复的。如果列作为主键，Unique属性应该设置为true

22.3.1　构建DataColumn

我们来继续前面的项目SimpleDataSet，并且演示怎么使用DataColumn为Inventory表建立列。因为CarID列是表的主键，对于这个DataColumn对象，我们将它设置为只读、唯一和不允许空值（通过ReadOnly、Unique和AllowDBNull属性）。接着，用FillDataSet()方法更新Program类，该方法用于建立4个DataColumn对象。注意该方法将DataSet对象作为它的唯一参数：

```
static void FillDataSet(DataSet ds)
{
  // 建立对应AutoLot数据库Inventory表真实字段的数据列
  DataColumn carIDColumn = new DataColumn("CarID", typeof(int));
  carIDColumn.Caption = "Car ID";
  carIDColumn.ReadOnly = true;
  carIDColumn.AllowDBNull = false;
  carIDColumn.Unique = true;

  DataColumn carMakeColumn = new DataColumn("Make", typeof(string));
  DataColumn carColorColumn = new DataColumn("Color", typeof(string));
  DataColumn carPetNameColumn = new DataColumn("PetName", typeof(string));
  carPetNameColumn.Caption = "Pet Name";
}
```

注意，在配置carIDColumn对象的时候，我们就为Caption属性赋了一个值。这个属性很有用，因为它能让我们定义一个用于显示的、和原先列名不一样的一个字符串值［要知道，数据库表中的列名

（如au_fname）更适合用于编程而不是显示（如Author First Name）]。这里基于相同的原因为PetName列设置标题，因为对于终端用户来说，Pet Name比PetName更好看。

22.3.2 启用自增列

可能会配置的DataColumn的另一个方面就是自增的能力。简单地说，自增列是为了保证为表添加了新行后，该列自动被分配值，该值基于当前的自增步进。当需要确保该列的值是不重复的时候非常有用（比如说主键）。

这个功能使用AutoIncrement、AutoIncrementSeed和AutoIncrementStep属性来实现。第二个属性指定列的起始值，第三个参数指定每次增加的步进值。看下面对carIDColumn的DataColumn改进：

```
static void FillDataSet(DataSet ds)
{
  DataColumn carIDColumn = new DataColumn("CarID", typeof(int));
  carIDColumn.ReadOnly = true;
  carIDColumn.Caption = "Car ID";
  carIDColumn.AllowDBNull = false;
  carIDColumn.Unique = true;
  carIDColumn.AutoIncrement = true;
  carIDColumn.AutoIncrementSeed = 0;
  carIDColumn.AutoIncrementStep = 1;
...
}
```

在这里，我们配置了carIDColumn对象来保证当表新增了一行后，列值自增1。因为我们设置了起始值为0，所以列值会是0、1、2、3等。

22.3.3 把DataColumn对象加入DataTable

DataColumn类型显然不能单独存在，需要加入到一个相关的DataTable中去。为了演示，我们创建一个新的DataTable对象（稍后会详细介绍），然后使用Columns属性加入列集合中的所有DataColumn对象，如下所示：

```
static void FillDataSet(DataSet ds):
{
...
  // 把DataColumn加入到DataTable
  DataTable inventoryTable = new DataTable("Inventory");
  inventoryTable.Columns.AddRange(new DataColumn[]
    { carIDColumn, carMakeColumn, carColorColumn, carPetNameColumn });
}
```

至此，DataTable对象有4个DataColumn对象，用来表示内存中Inventory表的架构。然而，表当前并没有数据，并且也不在DataSet维护的表集合中。我们稍后会处理这些不足，首先让我们通过DataRow对象来填充表。

22.4 使用 DataRow

我们知道，DataColumn对象的集合用来表示DataTable的结构。反之，DataRow对象的集合表示的是

表中的实际数据。因此，如果在AutoLot数据库的Inventory表中有20条记录，你就可以使用20个DataRow对象来呈现它们。表22-4列举了一些（但不是全部）DataRow类型的成员。

表22-4　DataRow类型的主要成员

成　　员	作　　用
HasErrors、GetColumnsInError()、GetColumnError()、ClearErrors()和RowError	HasErrors属性返回一个布尔值，指示该行是否存在错误。如果有错误，GetColumnsInError()可以用来获取包含错误的成员，GetColumnError()获取错误说明，而ClearError()用来清除该行的所有错误。RowErrors属性可以为错误设置一个自定义的文本说明
ItemArray	这个属性通过一个对象数组来获取或设置此行的所有列值
RowState	这个属性用来得到DataTable中当前DataRow的“状态”，值由RowState枚举定义（如行可以被标记为新的、已修改的、未改变的或已删除的
Table	这个属性用来获取包含该DataRow的DataTable
AcceptChanges()和RejectChanges()	这些方法提交或拒绝自上次调用AcceptChanges以来对该行进行的所有更改
BeginEdit()、EndEdit()和CancelEdit()	这些方法开始、结束和取消对某DataRow对象的编辑操作
Delete()	这个方法能标记该行为待删除，调用AcceptChanges()后移除该行
IsNull()	这个方法获取一个值，该值指示指定的列是否包含空值

使用DataRow和使用DataColumn有些不同，因为没有公共构造函数，所以你不能直接创建该类型的实例：

```
// 错误!没有公共构造函数
DataRow r = new DataRow();
```

正确的方法是从某个DataTable获得新的DataRow对象。比如，假设你想在Inventory表中插入两行，DataTable.NewRow()方法能让你获得该表的下一个“插槽”，然后通过类型索引器为每列赋值。我们可以指定赋值给DataColumn的字符串名或它的顺序位置（以0开始），下面是代码：

```
static void FillDataSet(DataSet ds)
{
...
  // 现在为Inventory表增加一些行
  DataRow carRow = inventoryTable.NewRow();
  carRow["Make"] = "BMW";
  carRow["Color"] = "Black";
  carRow["PetName"] = "Hamlet";
  inventoryTable.Rows.Add(carRow);

  carRow = inventoryTable.NewRow();
  // 第0列是自增ID字段，因此从1开始
  carRow[1] = "Saab";
  carRow[2] = "Red";
  carRow[3] = "Sea Breeze";
  inventoryTable.Rows.Add(carRow);
}
```

说明　如果传入DataRow索引器方法的是无效的列名或无效的顺序位置，我们就会收到运行时异常。

至此，已经有一个DataTable包含了两行。当然，我们可以重复这个过程，创建许多DataTable，定义架构和数据内容。在把inventoryTable对象插入DataSet对象中之前，让我们来看看所有重要的RowState属性。

22.4.1 RowState属性

当需要以编程方式指定表中的一组行被修改、新增等的时候，就会使用RowState属性。这个属性的值可以是DataRowState枚举的任何值，如表22-5所示。

表22-5 DataRowState枚举的值

值	作 用
Added	该行已添加到DataRowCollection中，AcceptChanges()尚未调用
Deleted	该行标记为通过DataRow的Delete()方法被删除，并且没有调用AcceptChanges()方法
Detached	该行已被创建，但不属于任何DataRowCollection。DataRow在以下情况下立即处于此状态：创建之后，添加到集合中之前或者从集合中移除之后
Modified	该行已被修改，AcceptChanges()尚未调用
Unchanged	该行自上次调用AcceptChanges()以来尚未更改

22

当以编程方式操作DataTable表中的行时，RowState属性会被自动设置作为一个示例。例如，向Program类新增一个方法，它操作本地DataRow对象，然后输出它的行状态，如下所示：

```
private static void ManipulateDataRowState()
{
  // 创建一个临时的DataTable用于测试
  DataTable temp = new DataTable("Temp");
  temp.Columns.Add(new DataColumn("TempColumn", typeof(int)));

  // RowState = Detached (i.e. not part of a DataTable yet)
  DataRow row = temp.NewRow();
  Console.WriteLine("After calling NewRow(): {0}", row.RowState);

  // RowState = Added
  temp.Rows.Add(row);
  Console.WriteLine("After calling Rows.Add(): {0}", row.RowState);

  // RowState = Added
  row["TempColumn"] = 10;
  Console.WriteLine("After first assignment: {0}", row.RowState);

  // RowState = Unchanged
  temp.AcceptChanges();
  Console.WriteLine("After calling AcceptChanges: {0}", row.RowState);

  // RowState = Modified
  row["TempColumn"] = 11;
  Console.WriteLine("After first assignment: {0}", row.RowState);

  // RowState = Deleted
  temp.Rows[0].Delete();
  Console.WriteLine("After calling Delete: {0}", row.RowState);
}
```

可见，ADO.NET DataRow非常“聪明”，它能记住操作后的状态。因此，包含DataRow的DataTable就能识别哪些行是添加的、修改过的或者删除的。当我们向数据库提交更新信息的时候，只有修改过的数据才会被提交，这是DataSet的一个主要特性。

22.4.2 DataRowVersion属性

除了通过RowState属性维护行的当前状态外，DataRow对象还通过DataRowVersion属性维护了它包含的数据的3个可能版本。在首次构建DataRow对象时，它仅包含一份数据，表示为“当前版本”。然而，当我们以编程方式操作DataRow对象（通过各种方法调用）后，就会引入其他版本的数据。具体而言，DataRowVersion可以设置为DataRowVersion枚举的任何相关值（如表22-6所示）。

表22-6　DataRowVersion枚举值

值	作　用
Current	表示行的当前值，即使在做出改变后
Default	DataRowState的默认值。对于Added、Modified或Deleted值的DataRowState，默认版本就是Current。对于Detached的DataRowState，版本就是Proposed
Original	表示首次插入DataRow的值，或AcceptChanges()最后一次被调用后的值
Proposed	表示调用BeginEdit()后当前正在被编辑的行的值

正如表22-6所示，DataRowVersion属性的值在很多情况下取决于DataRowState属性的值。之前说过，DataRowVersion属性在我们调用了DataRow（某种情况下是DataTable）的各种方法之后会悄悄改变。下面对可以影响行的DataRowVersion属性值的方法进行一些分析。

- 如果调用DataRow.BeginEdit()方法并且改变行的值，Current和Proposed值就可用了。
- 如果调用DataRow.CancelEdit()方法，就删除了Proposed值。
- 在调用DataRow.EndEdit()之后，Proposed值就变为Current值。
- 在调用DataRow.AcceptChanges()方法之后，Original值就和Current值一样了。在调用DataTable.AcceptChanges()后也会发生相同的转换。
- 在调用DataRow.RejectChanges()之后，Proposed值就被取消了，版本就变成了Current。

是的，这确实很复杂——主要是因为DataRow在任何时候都可能有也可能没有所有的版本（如果我们尝试获取当前没有跟踪的行版本就会收到运行时异常）。虽然很复杂，但是DataRow维护3份数据副本，构建允许用户改变值、改变主意之后回滚值或永久提交值的前端就变得很简单。在本章剩余部分中，我们会看到操作这些方法的各种示例。

22.5 使用DataTable

DataTable类型定义了许多成员，许多在名字和功能上和DataSet的差不多。表22-7列举了DataTable类型除了Rows和Columns外的一些主要属性。

表22-7 DataTable类型的主要成员

属 性	作 用
CaseSensitive	指示表中的字符串比较是否区分大小写，默认为false
ChildRelations	获取此DataTable的子关系的集合（如果有）
Constraints	获取由该表维护的约束的集合
Copy()	将某个DataTable的架构和数据复制到新实例中
DataSet	获取此表所属的DataSet（如果有）
DefaultView	获取可能包括筛选视图或游标位置的表的自定义视图
ParentRelations	获取该DataTable的父关系的集合
PrimaryKey	获取或设置充当数据表主键的列的数组
TableName	获取或设置表的名称，同样能通过构造参数指定这个属性

在当前例子中，让我们先来把DataTable的PrimaryKey属性设置为DataColumn对象carIDColumn。要知道，将DataColumn对象的集合赋给了PrimaryKey属性，表示多列键。然而在这里，我们只需要指定CarID列（是表中第一个位置），如下所示：

```
static void FillDataSet(DataSet ds)
{
...
  // 为这个表指定主键
  inventoryTable.PrimaryKey = new DataColumn[] { inventoryTable.Columns[0] };
}
```

22.5.1 将DataTable插入到DataSet中

做完这个后，DataTable的例子就结束了。最后一步就是使用Tables集合把DataTable对象插入到carsInventoryDS这个DataSet对象，如下所示：

```
static void FillDataSet(DataSet ds)
{
...
  // 最后，把我们的表加入到DataSet
  ds.Tables.Add(inventoryTable);
}
```

现在更新Main()方法以调用FillDataSet()，然后传递这个DataSet对象作为参数。接着将这个对象传递到一个叫PrintDataSet()的辅助方法（还没有写），如下所示：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with DataSets *****\n");
...
  FillDataSet(carsInventoryDS);
  PrintDataSet(carsInventoryDS);
  Console.ReadLine();
}
```

22.5.2　获取DataSet中的数据

这个PrintDataSet()方法通过ExtendedProperties集合迭代DataSet元数据和DataSet中的每一个DataTable，并且使用索引器输出列名和行值：

```
static void PrintDataSet(DataSet ds)
{
  // 输出DataSet名称和扩展属性
  Console.WriteLine("DataSet is named: {0}", ds.DataSetName);
  foreach (System.Collections.DictionaryEntry de in ds.ExtendedProperties)
  {
    Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
  }
  Console.WriteLine();

  // 输出每一张表
  foreach (DataTable dt in ds.Tables)
  {
    Console.WriteLine("=> {0} Table:", dt.TableName);

    // 输出列名
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Console.Write(dt.Columns[curCol].ColumnName + "\t");
    }
    Console.WriteLine("\n----------------------------------");

    // 输出DataTable
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
      for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
      }
      Console.WriteLine();
    }
  }
}
```

如果现在运行程序，就会看到下面的输出结果（时间戳和GUID）值当然会不同：

```
***** Fun with DataSets *****

DataSet is named: Car Inventory
Key = TimeStamp, Value = 1/22/2012 6:41:09 AM
Key = DataSetID, Value = 11c533ed-d1aa-4c82-96d4-b0f88893ab21
Key = Company, Value = Mikko's Hot Tub Super Store

=> Inventory Table:
CarID   Make    Color   PetName
----------------------------------
0       BMW     Black   Hamlet
1       Saab    Red     Sea Breeze
```

22.5.3 使用DataTableReader对象处理DataTable

如果你看过第21章的话，肯定注意到了使用连接层（如数据读取器对象）和断开连接层（如DataSet对象）处理数据的方式很不同。使用数据读取器一般会创建一个while循环，调用Read()方法并且使用一个索引器来取出名称/值对。在另一方面，DataSet处理一般会包含一系列迭代结构来挖掘其中的表、行和列（记得DataReader需要一个打开的数据库连接，以便从实际的数据库中读取数据）。

DataTable支持一个叫做CreateDataReader()的方法。这个方法允许使用像数据读取器一样的方式（数据读取器会从内存DataTable而不是实际的数据库中读取数据，因此这里不涉及数据库连接）来获取DataTable内的数据。这个方法的主要优势是，我们可以用一个模型来处理数据，而不去管使用哪个层的ADO.NET。为了演示，我们在Program类中创建一个新的名为PrintTable()的辅助方法，实现如下：

22

```
static void PrintTable(DataTable dt)
{
  // 得到DataTableReader类型
  DataTableReader dtReader = dt.CreateDataReader();

  // 像数据读取器一样操作DataTableReader
  while (dtReader.Read())
  {
    for (int i = 0; i < dtReader.FieldCount; i++)
    {
      Console.Write("{0}\t", dtReader.GetValue(i).ToString().Trim());
    }
    Console.WriteLine();
  }
  dtReader.Close();
}
```

注意，操作DataTableReader的方式和操作数据提供程序的数据读取器对象差不多。当想要从DataTable中快速取出数据时，DataTableReader是一个理想的解决方案，它不需要手动遍历DataTable内的行列集合。假定修改了前面的PrintDataSet()方法以调用PrintTable()方法，而没有遍历Rows和Columns集合：

```
static void PrintDataSet(DataSet ds)
{
  // 输出DataSet名字和扩展属性
  Console.WriteLine("DataSet is named: {0}", ds.DataSetName);
  foreach (System.Collections.DictionaryEntry de in ds.ExtendedProperties)
  {
    Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
  }
  Console.WriteLine();

  foreach (DataTable dt in ds.Tables)
  {
    Console.WriteLine("=> {0} Table:", dt.TableName);

    // 输出列名
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Console.Write(dt.Columns[curCol].ColumnName.Trim() + "\t");
    }
```

```
        Console.WriteLine("\n----------------------------------");

        // 调用新的辅助方法
        PrintTable(dt);
    }
}
```

此时运行应用程序，得到的输出结果与前面一样。唯一的不同是在内部访问DataTable的内容的方式。

22.5.4　序列化DataTable/DataSet对象为XML

DataSet和DataTable都支持WriteXml()和ReadXml()方法。WriteXml()允许把它们的内容持久化成XML文档形式的本地文件（包括所有从System.IO.Stream继承的类型）。ReadXml()允许从XML文档加载数据到DataSet（或者DataTable）。另外，DataSet和DataTable都支持WriteXmlSchema()和ReadXmlSchema()来保存和加载一个*.xsd文件。

为了做个测试，请修改Main()方法以调用最后一个辅助方法（DataSet是唯一参数）：

```
static void SaveAndLoadAsXml(DataSet carsInventoryDS)
{
  // 保存这个DataSet为XML
  carsInventoryDS.WriteXml("carsDataSet.xml");
  carsInventoryDS.WriteXmlSchema("carsDataSet.xsd");

  // 清除DataSet
  carsInventoryDS.Clear();

  // 从XML文件中加载DataSet
  carsInventoryDS.ReadXml("carsDataSet.xml");
}
```

如果打开carsDataSet.xml文件（它将位于你的项目的\bin\Debug文件夹下），你会看到表中的各个列都被编码成了XML元素：

```
<?xml version="1.0" standalone="yes"?>
<Car_x0020_Inventory>
  <Inventory>
    <CarID>0</CarID>
    <Make>BMW</Make>
    <Color>Black</Color>
    <PetName>Hamlet</PetName>
  </Inventory>
  <Inventory>
    <CarID>1</CarID>
    <Make>Saab</Make>
    <Color>Red</Color>
    <PetName>Sea Breeze</PetName>
  </Inventory>
</Car_x0020_Inventory>
```

在Visual Studio中双击生成的*.xsd文件（同样位于\bin\Debug文件夹），将打开IDE的XML架构编辑器（如图22-3所示）。

说明　第24章将介绍LINQ to XML API，它是在.NET平台下操纵XML数据的首选方式。

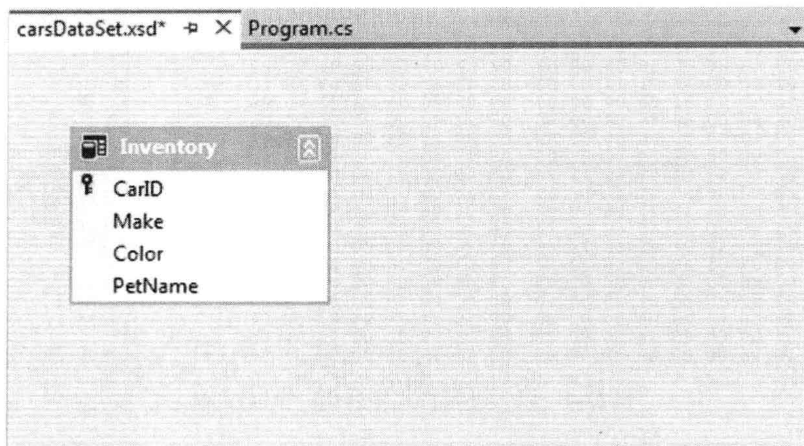


图22-3 Visual Studio的XSD编辑器

22.5.5 以二进制格式序列化DataTable/DataSet对象

还可以把DataSet（或单个DataTable）的内容以紧凑二进制格式进行持久化。如果DataSet对象需要跨越机器边界传递的话，这就特别有用（比如在分布式应用程序中）。XML数据表现的一个劣势就是其强描述性可能会导致大量的负担。

为了以二进制格式持久化DataTable或DataSet，只需要设置RemotingFormat属性为SerializationFormat.Binary。至此，我们就可以按照期望使用BinaryFormatter类型（见第20章）。考虑SimpleDataSet项目的最后一个方法（不要忘记导入System.IO和System.Runtime. Serialization. Formatters.Binary命名空间）：

```
static void SaveAndLoadAsBinary(DataSet carsInventoryDS)
{
  // 设置二进制序列化标记
  carsInventoryDS.RemotingFormat = SerializationFormat.Binary;

  // 以二进制格式保存DataSet
  FileStream fs = new FileStream("BinaryCars.bin", FileMode.Create);
  BinaryFormatter bFormat = new BinaryFormatter();
  bFormat.Serialize(fs, carsInventoryDS);
  fs.Close();

  // 清空DataSet
  carsInventoryDS.Clear();

  // 从二进制文件加载DataSet
  fs = new FileStream("BinaryCars.bin", FileMode.Open);
  DataSet data = (DataSet)bFormat.Deserialize(fs);
}
```

如果从Main()中调用该方法，就会在bin\Debug文件夹中找到*.bin文件。图22-4显示了BinaryCars.bin文件的内容。

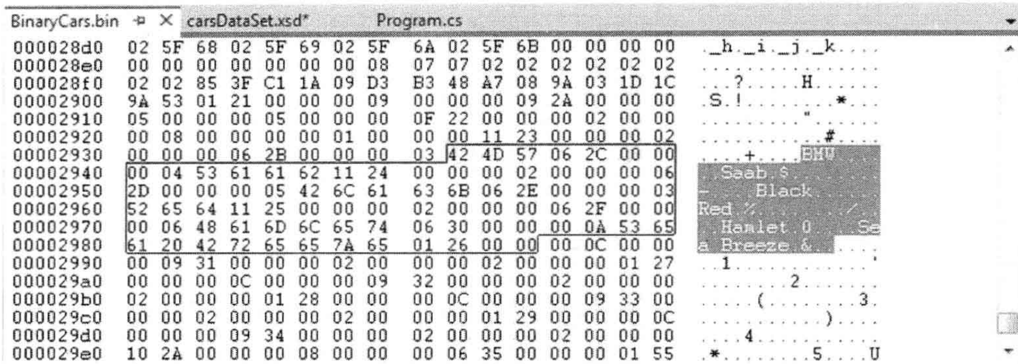


图22-4　以二进制格式序列化一个DataSet

源代码　SimpleDataSet应用程序的源代码位于Chapter 22子目录下。

22.6　将DataTable对象绑定到用户界面

至此，我们已经研究了如何使用ADO.NET的继承对象模型手动创建、合成和迭代DataSet对象的内容。虽然理解如何实现很重要，但是.NET平台有许多API提供了自动将数据绑定到用户界面元素的能力。

例如，.NET的原始GUI工具包Windows Forms支持一个叫DataGridView的控件，它内建了使用几行代码来显示DataSet或DataTable对象内容的能力。ASP.NET（.NET Web开发API）和WPF API（.NET 3.0引入的增强的GUI API）也支持以相似形式实现数据绑定。本书后面将学习如何绑定数据到WPF和ASP.NET GUI元素。但是在本章中，你将使用Windows Forms，因为它是相当简单的编程模型。

说明　下一个示例假设你已经具备使用Windows Forms来构建GUI的经验。如果不是这样的话，你最好先学习一下解决方案，或者在阅读附录A后返回到这部分内容。

下一个任务就是构建一个Windows Forms应用程序，在用户界面上显示DataTable对象的内容。然后，我们还会研究如何过滤和改变表数据，并且我们会了解DataView对象的作用。

首先，创建一个叫做WindowsFormsDataBinding的全新Windows Forms项目工作区。使用Solution Explorer将初始的Form1.cs重命名为更合适的MainForm.cs。然后，使用Visual Studio工具箱将Data选项卡上的DataGridView控件（通过Properties窗口的Name属性重命名为carInventoryGridView）拖曳到设计器界面上。你可能会注意到，当第一次向设计器添加DataGridView时，激活了一个允许连接到物理数据源的上下文菜单。但是现在忽略设计器的这个方面，因为我们会通过编程绑定DataTable对象。最后，为了显示信息，向设计器增加一个描述性的Label。图22-5演示了可能的外观。

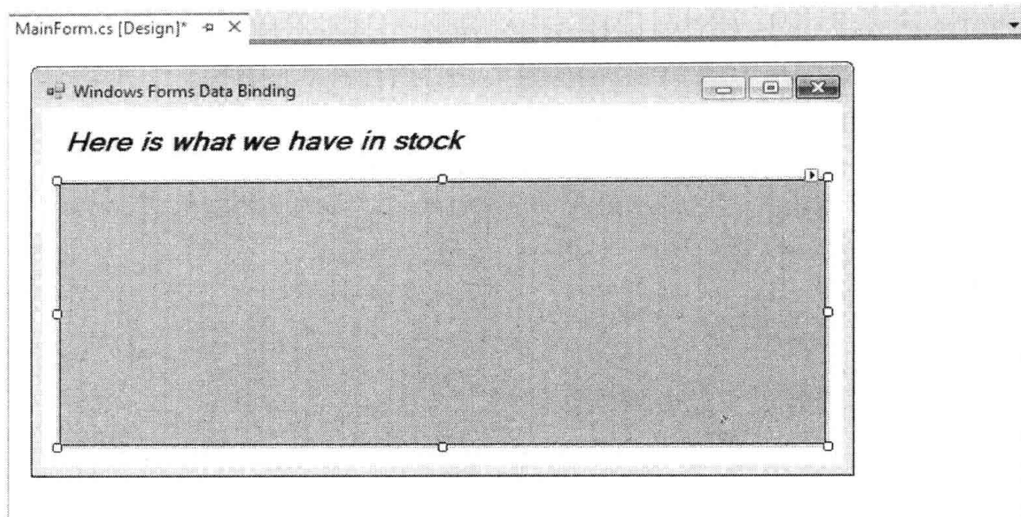


图22-5 Windows Forms应用程序最初的GUI

22.6.1 从泛型List<T>合成DataTable

和之前的SimpleDataSet示例相似，WindowsFormsDataBinding应用程序会构建一个DataTable来包含一组表示各列各行数据的DataColumn。然而，现在我们会使用泛型List<T>成员变量填充行。首先，为我们的项目插入一个叫做Car的新的C#类，定义如下：

```
public class Car
{
  public int ID { get; set; }
  public string PetName { get; set; }
  public string Make { get; set; }
  public string Color { get; set; }
}
```

现在，在默认构造函数中，使用一个叫listCars的List<T>成员变量来填充一组Car对象，如下所示：

```
public partial class MainForm : Form
{
  // Car对象的集合
  List<Car> listCars = null;

  public MainForm()
  {
    InitializeComponent();

    // 填充一些汽车
    listCars = new List<Car>
    {
      new Car { ID = 100, PetName = "Chucky", Make = "BMW", Color = "Green" },
      new Car { ID = 101, PetName = "Tiny", Make = "Yugo", Color = "White" },
      new Car { ID = 102, PetName = "Ami", Make = "Jeep", Color = "Tan" },
      new Car { ID = 103, PetName = "Pain Inducer", Make = "Caravan",
                Color = "Pink" },
```

```
        new Car { ID = 104, PetName = "Fred", Make = "BMW", Color = "Green" },
        new Car { ID = 105, PetName = "Sidd", Make = "BMW", Color = "Black" },
        new Car { ID = 106, PetName = "Mel", Make = "Firebird", Color = "Red" },
        new Car { ID = 107, PetName = "Sarah", Make = "Colt", Color = "Black" },
      };
    }
  }
```

然后，向MainForm类增加一个叫inventoryTable的DataTable类型成员变量，如下所示：

```
public partial class MainForm : Form
{
  // Car对象的集合
  List<Car> listCars = null;

  // 库存信息
  DataTable inventoryTable = new DataTable();
...
}
```

为我们的类增加一个叫CreateDataTable()的辅助函数，并且在MainForm类的默认构造函数中调用这个方法：

```
private void CreateDataTable()
{
  // 创建表构架
  DataColumn carIDColumn = new DataColumn("ID", typeof(int));
  DataColumn carMakeColumn = new DataColumn("Make", typeof(string));
  DataColumn carColorColumn = new DataColumn("Color", typeof(string));
  DataColumn carPetNameColumn = new DataColumn("PetName", typeof(string));
  carPetNameColumn.Caption = "Pet Name";
  inventoryTable.Columns.AddRange(new DataColumn[] { carIDColumn,
    carMakeColumn, carColorColumn, carPetNameColumn });

  // 迭代数组列表List<T>来创建行
  foreach (Car c in listCars)
  {
    DataRow newRow = inventoryTable.NewRow();
    newRow["ID"] = c.ID;
    newRow["Make"] = c.Make;
    newRow["Color"] = c.Color;
    newRow["PetName"] = c.PetName;
    inventoryTable.Rows.Add(newRow);
  }

  // 把DataTable绑定到carInventoryGridView
  carInventoryGridView.DataSource = inventoryTable;
}
```

这个方法的实现首先通过创建4个DataColumn对象来创建DataTable的架构（为了简单，我们没有增加自动递增的ID字段或把它设为主键），然后我们把它们加入到DataTable成员变量中。然后使用foreach迭代结构和原生ADO.NET对象模型将行数据从List<Car>字段映射到DataTable。

然而，注意CreateDataTable()方法中最后一个代码语句将inventoryTable赋值给了DataGridView对象的DataSource属性。只需要设置这个属性就可以绑定DataTable到DataGridView对象。GUI控件会在内部读取行列集合，和我们在SimpleDataSet示例中处理的PrintDataSet()方法很相似。至此就可以运行应用程序了，我们将看到DataGridView控件中的DataTable，如图22-6所示。

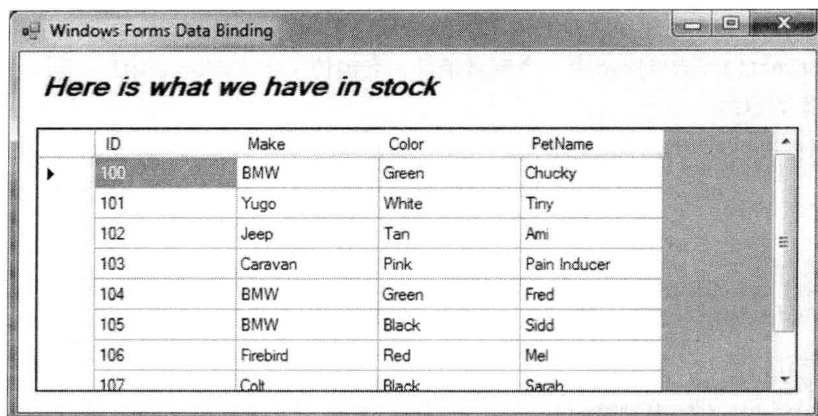


图22-6 绑定DataTable到Windows Forms DataGridView

22.6.2 从DataTable中删除行

现在如果想修改图形界面，使用户能删除绑定到DataGridView的DateTable中的行，一个办法就是调用DataRow对象的Delete()方法来表明这行即将删除。只需指定待删除行的索引（或者DataRow对象）。为了使用户能指定要删除的行，将一个TextBox（名为txtRowRemove）和一个Button控件（名为btnRemoveRow）添加到当前设计器。如图22-7显示了一个可能的UI更新（注意本例在GroupBox控件中包括两个控件，以便强调它们是如何关联的）。

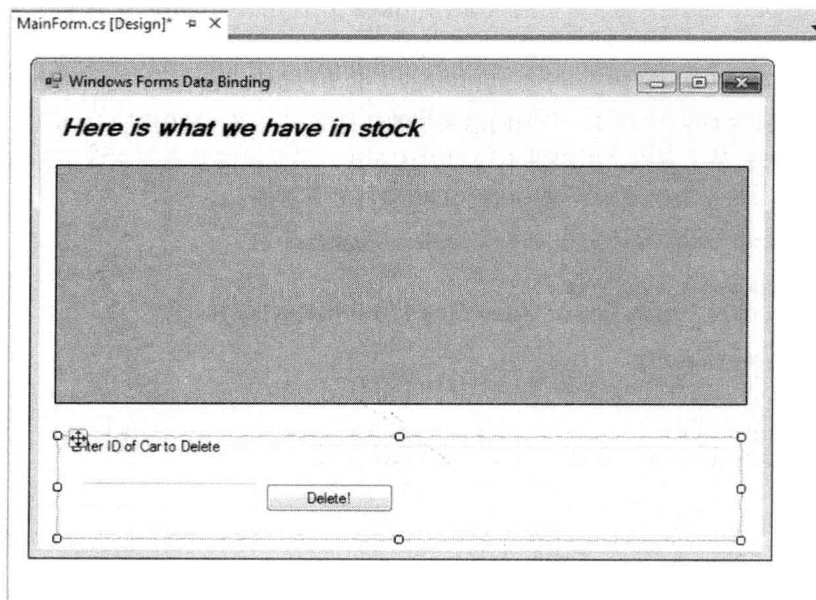


图22-7 修改UI以允许从DataTable移除行

新Button的Click事件处理程序根据汽车的ID，从位于内存的DataTable中移除了用户指定的行。DataTable类的Select()方法可以指定一个搜索条件，它模仿了正常的SQL语法。返回值为匹配搜索条件的DataRow对象的数组：

```
// 从DataRowCollection删除这行
private void btnRemoveCar_Click (object sender, EventArgs e)
{
  try
  {
    // 找到要删除的行
    DataRow[] rowToDelete = inventoryTable.Select(
      string.Format("ID={0}", int.Parse(txtCarToRemove.Text)));

    // 删除它
    rowToDelete[0].Delete();
    inventoryTable.AcceptChanges();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

现在可以运行应用程序并且指定要从DataTable中删除的汽车ID。如果从DataTable移除DataRow对象的话，我们会发现网格的UI立即更新了，因为它绑定了到DataTable对象的状态。

22.6.3　根据筛选条件选择行

很多数据相关的应用程序需要通过指定某种过滤标准来查看DataTable数据的一个小集合。例如，可以只看到内存中DataTable中某个牌子的汽车（如BMW）。使用DataTable类的Select()方法，我们就可以找到要删除的行，然而为了显示，也可以用这个方法选择记录的子集。

再次修改图形界面，这次使用一个新的TextBox（名为txtMakeToView）和一个新的Button（名为btnDisplayMakes）能允许用户填写一个他们感兴趣的品牌的字符串（如图22-8所示）。

Select()方法被重载了多次，以提供不同的筛选方式。最简单的就是用包含条件操作符的字符串作为参数值来指定。首先观察下面新建按钮的Click事件处理程序：

```
private void btnDisplayMakes_Click(object sender, EventArgs e)
{
  // 根据用户输入的内容构建过滤器
  string filterStr = string.Format("Make= '{0}'", txtMakeToView.Text);

  // 查找符合条件的所有记录
  DataRow[] makes = inventoryTable.Select(filterStr);

  if (makes.Length == 0)
    MessageBox.Show("Sorry, no cars...", "Selection error!");
  else
  {
    string strMake = "";
    for (int i = 0; i < makes.Length; i++)
    {
      // 从当前行开始，获取PetName的值
      strMake += makes[i]["PetName"] + "\n";
    }
```

```
      // 显示消息框中所有符合条件的记录
      MessageBox.Show(strMake,
        string.Format("We have {0}s named:", txtMakeToView.Text));
    }
  }
```

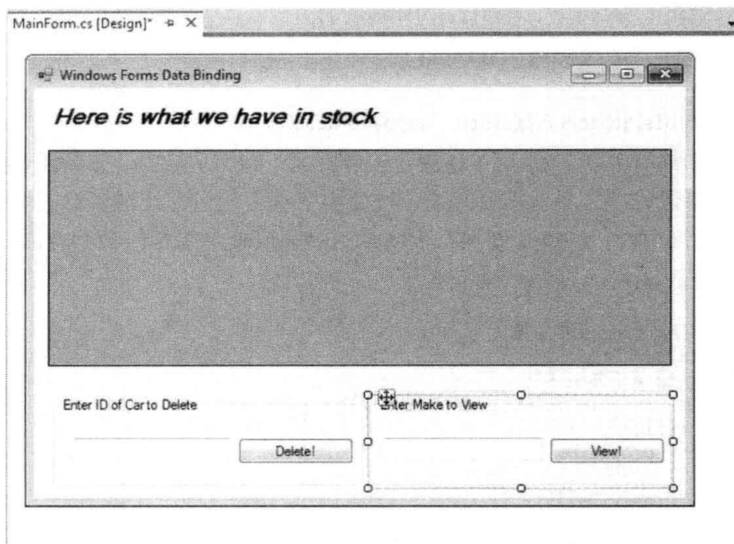


图22-8　更新UI以启用行筛选

在这里，我们做的这个简单的过滤器是以TextBox的值为基础的。如果你填写BMW，过滤参数就是 Make='BMW'。当使用Select()方法提交这个条件时，你会得到一个符合这个条件的DataRow类型数组（如图22-9所示）。

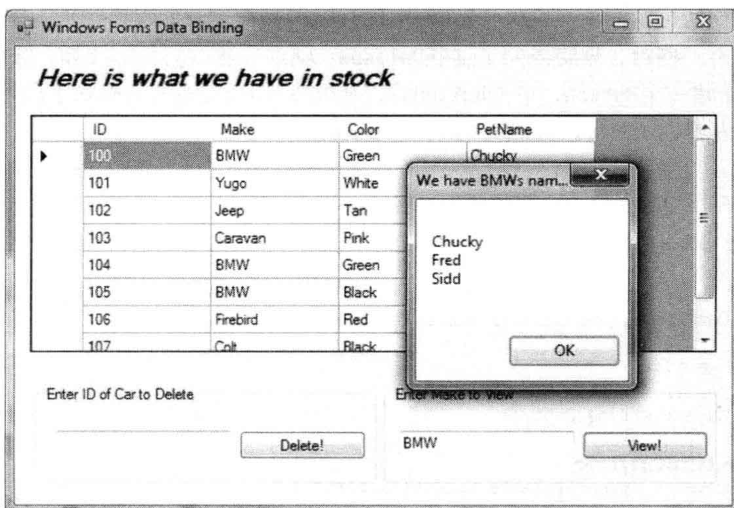


图22-9　显示过滤后的信息

可见，过滤参数是标准的SQL语法。假设你想按照PetName字段的字母顺序得到前面的Select()执行结果，对于SQL语句而言，就是按照PetName列进行排序。幸好Select()方法已被重载，有一个相应的标准能实现这个功能，下面是代码：

```
// 按照PetName排序
makes = inventoryTable.Select(filterStr, "PetName");
```

如果你想要结果倒序，像这样调用Select()：

```
// 以倒序返回结果
makes = inventoryTable.Select(filterStr, "PetName DESC");
```

通常，这个排序字符串由列名和ASC（正序）或者DESC（倒序）组成。如果需要的话，可以用逗号分割字段来按照多个字段排序。最后需要知道，过滤器字符串能包含任意多的关系操作。例如，当你想查找所有ID大于5的汽车时，该怎么做呢？下面是一个完成这个工作的辅助方法：

```
private void ShowCarsWithIdGreaterThanFive()
{
  // 现在显示所有ID大于5的汽车名字
  DataRow[] properIDs;
  string newFilterStr = "ID > 5";
  properIDs = inventoryTable.Select(newFilterStr);
  string strIDs = null;
  for(int i = 0; i < properIDs.Length; i++)
  {
    DataRow temp = properIDs[i];
    strIDs += temp["PetName"]
          + " is ID " + temp["ID"] + "\n";
  }
  MessageBox.Show(strIDs, "Pet names of cars where ID > 5");
}
```

22.6.4　在DataTable中更新行

最后需要了解如何用新的值更新DataTable中现有行的操作。一个方法就是先使用Select()方法按照一定的标准筛选出一些行，只要得到了这些DataRow，就逐一修改它们。比如，假设现在可以在表单btnChangeMakes上新建一个Button，单击Button后，从DataTable搜索所有Make字段等于BMW的行，然后逐一修改Make字段内容，从BMW改为Yugo，如下所示：

```
// 用过滤器找到想编辑的行
private void btnChangeMakes_Click(object sender, EventArgs e)
{
  // 确认用户是否改变主意
  if (DialogResult.Yes ==
    MessageBox.Show("Are you sure?? BMWs are much nicer than Yugos!",
      "Please Confirm!", MessageBoxButtons.YesNo))
  {
    // 建立一个过滤器
    string filterStr = "Make='BMW'";
    string strMake = string.Empty;

    // 找到所有匹配过滤器的行
    DataRow[] makes = inventoryTable.Select(filterStr);

    // 把所有的“Beemers”修改成“Yugos”
```

```
        for (int i = 0; i < makes.Length; i++)
        {
          makes[i]["Make"] = "Yugo";
        }
      }
    }
```

22.6.5 使用DataView类型

在数据库术语中，视图对象是一个表（或者一组表）自定义的表现形式。比如说，使用微软SQL Server可以为现在的Inventory表建立一个视图，返回一个只有某种特定颜色的汽车的表。在ADO.NET中，DataView类型允许以编程方式从DataTable中提取一组数据到独立的对象。

为同一个表建立多个视图最大的好处就是允许绑定它们到不同的GUI部件（比如DataGridView）。比如说，一个DataGridView绑定到DataView，显示Inventory表中的所有数据，另外一个则被配置成只显示绿色的车型。

为了演示，我们先为当前的用户界面另外增加一个名为dataGridYugosView的DataGridView和一个描述性Label。接着，定义一个DataView类型的变量yugoOnlyView：

```
public partial class MainForm : Form
{
  // DataTable的视图
  DataView yugosOnlyView;
...
}
```

现在，在程序默认的构造方法内新建一个叫做CreateDataView()的辅助方法，并且在DataTable建立后调用，代码如下：

```
public MainForm()
{
...
  // 建立一个数据表
  CreateDataTable();

  // 建立一个视图
  CreateDataView();
}
```

下面是这个新辅助方法的实现。需要注意的是，我们把DataTable作为构造参数传入DataView，用来建立自定义数据行集合：

```
private void CreateDataView()
{
  // 设置用来构建这个视图的表
  yugosOnlyView = new DataView(inventoryTable);

  // 用过滤器配置这个视图
  yugosOnlyView.RowFilter = "Make = 'Yugo'";

  // 绑定到新网格
  dataGridYugosView.DataSource = yugosOnlyView;
}
```

可见，DataView类支持一个叫做RowFilter的属性，用来设置表示筛选标准的字符串，以获得匹配的行。一旦建立了视图，相应地设置网格的DataSource属性。图22-10显示了完整的程序效果。

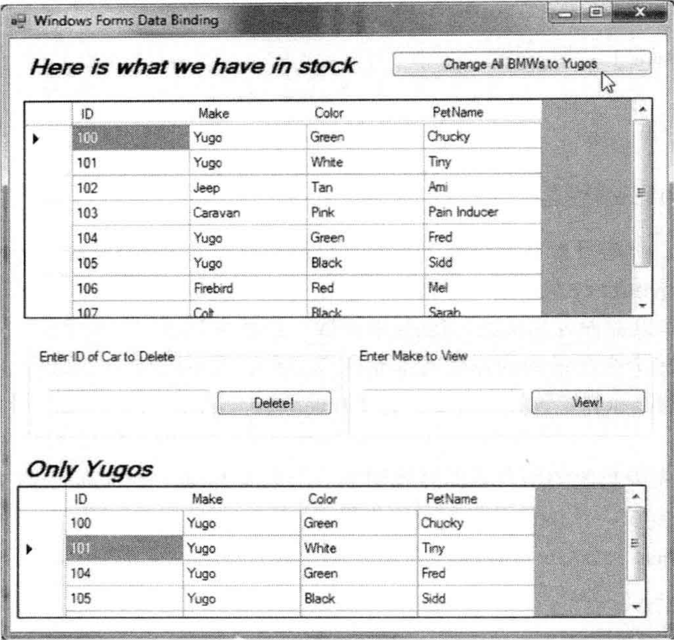


图22-10　显示数据的唯一视图

源代码　WindowsFormsDataBinding项目的源代码位于Chapter 22子目录下。

22.7　使用数据适配器

现在知道了如何操作ADO.NET DataSet进行输入输出，下面介绍一下数据适配器对象。数据适配器类用来向DataSet填充DataTable并把修改后的DataTable返回数据库处理。表22-8列举了一些DbDataAdapter基类的核心成员。这是每个数据适配器对象的公共父对象（例如SqlDataAdapter和OdbcDataAdapter）。

表22-8　DbDataAdapter类的核心成员

成　　员	作　　用
Fill()	执行SQL SELECT命令（由SelectCommand属性指定）查询数据库并将数据加载到DataTable中
SelectCommand、InsertCommand、UpdateCommand和DeleteCommand	建立用于Fill()和Update()方法的由数据库执行的SQL命令
Update()	执行SQL INSERT、UPDATE和DELETE命令（由InsertCommand、UpdateCommand和DeleteCommand属性指定来持久化DataTable修改到数据库中

注意数据适配器定义了4个属性：SelectCommand、InsertCommand、UpdateCommand和DeleteCommand。如果我们为某个数据提供程序创建数据适配器对象（如SqlDataAdapter），我们就可以传入SelectCommand命令对象使用的表示命令文本的字符串类型。

如果4个命令行对象都被正确配置了，我们就可以调用Fill()方法来获取DataSet（也可以选择一个DataTable）。这可以通过让数据适配器执行SelectCommand属性指定的SQL SELECT语句来实现。

同样，如果我们希望把修改后的DataSet（或DataTable）对象传回数据库来处理，我们就可以调用Update()方法，它会根据DataTable中每一行的状态来使用其他命令行对象（稍后我们会详细介绍）。

使用数据适配器对象一个奇怪的方面是我们从来不需要打开或关闭对数据库的连接，因为对数据库的基础连接会自动管理。然而，我们仍然需要为数据适配器提供有效的连接对象或连接字符串（在内部会用于构建连接对象）来通知数据适配器我们希望和哪个数据库进行通信。

说明 数据适配器在本质上是不可知的。你可以在运行时传入不同的连接对象和命令对象，也可以从各种各样的数据库中获取数据。例如，DataSet可以包含从SQL Server、Oracle、MySQL等数据库提供程序中得到的表数据。

22.7.1 一个简单的数据适配器示例

下一步就是向第21章中创建的数据访问库程序集AutoLotDAL.dll新增功能。首先创建一个简单的示例，它会使用ADO.NET数据适配器为DataSet填充一个表。

新建一个叫FillDataSetUsingSqlDataAdapter的控制台应用程序，并且把System.Data和System.Data.SqlClient命名空间导入到我们最初的C#代码文件中。现在，如下更新Main()方法（根据在第21章创建AutoLot数据库的方式，可能需要修改连接字符串）：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Data Adapters *****\n");

  // 硬编码的连接字符串
  string cnStr = "Integrated Security = SSPI;Initial Catalog=AutoLot;" +
    @"Data Source=(local)\SQLEXPRESS";

  // 调用者创建DataSet对象
  DataSet ds = new DataSet("AutoLot");

  // 告知适配器的Select命令文本和连接字符串
  SqlDataAdapter dAdapt =
    new SqlDataAdapter("Select * From Inventory", cnStr);

  // 为DataSet填充一个叫Inventory的新表
  dAdapt.Fill(ds, "Inventory");

  // 显示DataSet的内容，使用本章前面创建的辅助方法
  PrintDataSet(ds);
  Console.ReadLine();
}
```

注意，数据适配器已经通过指定映射到SQL SELECT语句的字符串文本来构建。这个值会被用来在内部构建命令对象，之后可以通过SelectCommand属性来获取这个命令对象。

接着，注意调用者需要自己来创建DataSet类型的实例，它被传入到Fill()方法。Fill()方法接受可选的第二个参数，即一个字符串，用来设置新DataTable的TableName属性（如果我们不指定表名的话，数据适配器会将表命名为Table）。虽然大多数情况下赋给DataTable的名字会和关系数据库中的物理表名字一样，但这不是必需的。

说明　Fill()方法返回一个整数来表示由SQL查询返回的行数。

最后，注意在Main()方法中我们没有显式打开或关闭对数据库的连接。某个数据适配器的Fill()方法已经被预编程，它将在从Fill()方法返回之前打开然后关闭基础连接。因此，当我们把DataSet传入PrintDataSet()方法时（本章之前实现的），只是操作断开连接的数据的本地副本，而不会发生获取数据的往返过程。

22.7.2　映射数据库名称为友好名称

前面提到过数据库管理员（DBA）往往会用简短的而不是对于终端用户来说友好的名称来作为表名和列名（如au_id、au_fname或au_lname）。幸好通过TableMappings属性能访问数据适配器对象保持的System.Data.Common.DataTableMapping类型的一个内部强命名的集合（DataTableMappingCollection）。

如果这么做的话，可以操作这个集合来告诉DataTable哪些是要在显示内容的时候使用“友好名”的。比如，假设你想在显示的时候把数据库名Inventory映射到Current Inventory，显示CarID列名为Car ID（注意中间的空格），显示PetName列名为Name of Car。要实现这个，需要在调用数据适配器的Fill()方法以前加入下面的代码（确信已经使用了System.Data.Common命名空间来获得DataTableMapping类型的定义）：

```
static void Main(string[] args)
{
...
  // 现在映射列名到一个用户友好的名字
  DataTableMapping custMap =
    dAdapt.TableMappings.Add("Inventory", "Current Inventory");
  custMap.ColumnMappings.Add("CarID", "Car ID");
  custMap.ColumnMappings.Add("PetName", "Name of Car");
  dAdapt.Fill(ds, "Inventory");
...
}
```

如果再次运行这个程序的话，会发现PrintDataSet()方法现在为DataTable和DataRow对象显示“友好名”而不是数据库结构的名字：

```
***** Fun with Data Adapters *****

DataSet is named: AutoLot
```

```
=> Current Inventory Table:
Car ID  Make    Color   Name of Car
----------------------------------
83      Ford    Rust    Rusty
107     Ford    Red     Snake
678     Yugo    Green   Clunker
904     VW      Black   Hank
1000    BMW     Black   Bimmer
1001    BMW     Tan     Daisy
1992    Saab    Pink    Pinkey
2003    Yugo    Rust    Mel
```

源代码 FillDataSetUsingSqlDataAdapter项目的源代码位于Chapter 22子目录下。

22.8 向 AutoLotDAL.dll 添加断开连接功能

为了演示使用数据适配器将DataTable中的修改发回到数据库处理的过程，我们现在更新第21章中创建的AutoLotDAL.dll程序集来包含新的命名空间AutoLotDisconnectedLayer。这个命名空间会包含一个叫InventoryDALDisLayer的新类，它会使用数据适配器来和DataTable交互。

一个比较好的方法是，将第21章创建的AutoLotDAL项目所在的文件夹整个复制到硬盘中新的位置，并将文件夹重命名为AutoLotDAL(Version Two)。现在使用Visual Studio，激活File→Open Project/Solution...菜单选项，打开AutoLotDAL (Version Two)文件夹下的AutoLotDAL.sln文件。

22.8.1 定义初始类类型

通过Project→Add Class菜单项插入一个叫InventoryDALDisLayer的新类，并且确保在新的代码文件中有一个公共类类型。

修改包括这个类的命名空间为AutoLotDisconnetedLayer并且导入System.Data和System.Data.SqlClient命名空间。

和与连接相关的InventoryDAL类型不同的是，新的类不需要提供自定义打开/关闭方法，因为数据适配器会自动处理这些细节。

首先，增加一个自定义构造函数来设置一个表示连接字符串的私有string变量。同样，定义一个私有的SqlDataAdapter成员变量，我们会通过ConfigureAdapter()辅助方法（还没有创建）来配置，它接受SqlDataAdapter输出参数：

```
namespace AutoLotDisconnectedLayer
{
  public class InventoryDALDisLayer
  {
  // 字段数据
    private string cnString = string.Empty;
    private SqlDataAdapter dAdapt = null;

    public InventoryDALDisLayer(string connectionString)
```

```
        {
          cnString = connectionString;

          // 配置SqlDataAdapter
          ConfigureAdapter(out dAdapt);
        }
      }
    }
```

22.8.2　使用SqlCommandBuilder来配置数据适配器

使用数据适配器来修改DataSet中的表时，第一件事情就是为UpdateCommand、DeleteCommand以及InsertCommand属性赋予有效的命令对象（在这样做之前，这些属性都会返回null引用）。

为InsertCommand、UpdateCommand和DeleteCommand属性手工配置命令对象需要大量的代码，特别是在我们使用参数化查询的时候。第21章中说过，参数化查询允许我们使用一组参数对象构建SQL语句。因此，如果希望自己走这条路的话，可以手动创建3个SqlCommand对象来实现ConfigureAdapter()，每一个对象都包含一组SqlParameter对象。然后，我们就可以把这些对象设置为适配器的UpdateCommand、DeleteCommand和InsertCommand属性。

值得庆幸的是，Visual Studio 提供了许多设计器工具来为我们处理这些没有技术含量而又枯燥的工作。这些设计器因所使用的API不同（如Windows Forms、WPF、ASP.NET）而有所区别，但整体的功能是类似的。使用这些设计器的示例将贯穿本书，包括本章稍后将使用的Windows Forms设计器。

不必编写大量代码语句来完整配置数据适配器，让我们按如下代码实现ConfigureAdapter()来精简大量的工作：

```
private void ConfigureAdapter(out SqlDataAdapter dAdapt)
{
  // 创建适配器并且设置SelectCommand
  dAdapt = new SqlDataAdapter("Select * From Inventory", cnString);

  // 使用SqlCommandBuilder在运行时动态获取其余的命令对象
  SqlCommandBuilder builder = new SqlCommandBuilder(dAdapt);
}
```

为了帮助简化数据适配器对象的构建，微软提供的每一个ADO.NET数据提供程序都提供了一个命令生成器类型。SqlCommandBuilder类型根据最初的SelectCommand自动生成包含在SqlDataAdapter的InsertCommand、UpdateCommand和DeleteCommand属性中的值。很明显，好处是我们不需要手动构建所有的SqlCommand和SqlParameter类型。

这个时候一个很明显的问题是命令生成器如何能即时构建这些SQL命令对象。简洁的答案就是利用元数据。在运行时，如果我们调用数据提供程序的Update()方法，相关命令生成器会读取数据库的构架数据来自动生成底层的插入、删除和更新命令对象。

显然，这样做的话会产生对远程数据库的额外往返，因此在一个应用程序中多次使用SqlCommandBuilder的话，肯定会有损性能。在这里，我们通过在构造InventoryDALDisLayer对象的时候调用ConfigureAdapter()方法来最小化负面效果，这样配置后的SqlDataApter就会在整个对象的生命周期中保留下来。

在之前的代码中，除了作为构造函数参数传入数据适配器以外，我们并没有使用命令生成器对象（在这里的SqlCommandBuilder）。可能看上去很奇怪，这就是我们需要做的所有工作吗？在背后，这个类型会使用其余的命令对象配置数据适配器。

现在，虽然你可能喜欢上了这个“空手套白狼”的主意，但是要知道命令生成器有很多关键的限制。具体而言，命令生成器只能在如下条件是true的时候才自动生成用于数据适配器的SQL命令。

- SQL SELECT命令必须和单个表交互（比如没有联接）。
- 这是一个必须有主键的表。
- 这个表必须有表示主键的列并且必须包含在我们的SQL SELECT语句中。

根据我们构建的AutoLot数据库，这些限制都没有问题，然而，在一个工业强度更大的数据库中，我们需要考虑这个类型是否有用（如果没有用的话，Visual Studio会自动生成大量需要的代码，在本章后面我们会看到）。

22

22.8.3 实现GetAllInventory()

既然数据适配器已经准备好了，新类类型的第一个方法只需要使用SqlDataAdapter对象的Fill()方法来获取表示AutoLot数据库中Inventory表所有记录的DataTable，如下所示：

```
public DataTable GetAllInventory()
{
  DataTable inv = new DataTable("Inventory");
  dAdapt.Fill(inv);
  return inv;
}
```

22.8.4 实现UpdateInventory()

UpdateInventory()方法很简单：

```
public void UpdateInventory(DataTable modifiedTable)
{
  dAdapt.Update(modifiedTable);
}
```

在这里，数据适配器对象会检查传入DataTable每行的RowState值。根据这个值（RowState.Added、RowState.Deleted、RowState.Modified）在后台使用正确的命令对象。

22.8.5 设置版本号

至此，数据访问库第二个版本的逻辑部分已经全部完成。你可以将该库的版本号设为2.0.0.0，尽管这不是必需的，但这样做是一个很好的习惯。如第14章所述，要修改.NET程序集的版本，可以双击Solution Explorer中的Properties节点，然后单击Application选项卡中的Assembly Information...按钮。在弹出的对话框中，将Assembly Version的Major号（主版本号）设置为2（详细内容参考第14章）。完成之后，重新编译应用程序以更新程序集清单。

源代码 AutoLotDAL（Version Two）项目的源代码位于Chapter 22子目录下。

22.8.6 测试非连接的功能

这时我们可以构建一个前端来测试新的InventoryDALDisLayer类。我们仍然使用Windows Forms API在图形用户界面上显示数据。新建一个Windows Forms应用程序InventoryDALDisconnectedGUI，使用Solution Explorer将初始的Form1.cs文件改为MainForm.cs，然后添加对新的AutoLotDAL.dll程序集的引用（要确保所添加的是2.0.0.0版），并引入下面的命名空间：

```
using AutoLotDisconnectedLayer;
```

表单的设计包含一个Label、DataGridView（叫inventoryGrid）和Button控件（叫btnUpdateInventory，我们将它配置为处理Click事件）。下面是表单的定义：

```
public partial class MainForm : Form
{
  InventoryDALDisLayer dal = null;

  public MainForm()
  {
    InitializeComponent();

    string cnStr =
    @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot;" +
     "Integrated Security=True;Pooling=False";

    // 创建数据访问对象
    dal = new InventoryDALDisLayer(cnStr);

    // 填充网格
    inventoryGrid.DataSource = dal.GetAllInventory();
  }

  private void btnUpdateInventory_Click(object sender, EventArgs e)
  {
    // 从网格获取修改后的数据
    DataTable changedDT = (DataTable)inventoryGrid.DataSource;

    try
    {
      // 提交我们的改动
      dal.UpdateInventory(changedDT);
    }
    catch(Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
  }
}
```

创建了InventoryDALDisLayer对象之后，将从GetAllInventory()返回的DataTable绑定到DataGridView对象。如果用户单击了Update按钮，我们就从网格（通过DataSource属性）提取修改后的DataTable并且把它传入UpdateInventory()方法。

就这么简单！运行程序后，增加一组新行到网格上并且更新/删除另外一些。如果单击Button控件，就会看到这些改动持久化到了AutoLot数据库中。

源代码 更新后的InventoryDALDisconnectedGUI项目的源代码位于Chapter 22子目录下。

22.9 多表 DataSet 对象和数据关系

当前，本章所有涉及DataSet的例子都只包含一个DataTable对象。然而，当一个DataSet对象包含数个关联的DataTable的时候才是发挥断开式访问强大力量的时刻。既然这样，我们就能把任意多的DataRelation对象插入到DataSet的DataRelation集合中去，来表示数据表之间的依赖关系。使用这些对象，客户端就能自由使用这些表，避免了不必要的网络流量。

说明 这个示例把数据访问逻辑隔离到了一个新的Windows Forms项目(而不是更新AutoLotDAL.dll)来处理Customers和Orders表。然而，对于产品级别的应用程序来说，不推荐把UI和数据访问逻辑混合在一起。本章最后的示例利用了各种数据库设计工具来解耦UI和数据逻辑代码。

我们新建一个叫做MultitabledDataSet的WindowsForms应用程序。用户界面非常简单(注意我将原始的Form1.cs改名为MainForm.cs)。在图22-11中，你能看到3个DataGridView部件(dataGridViewInventory、dataGridViewCustomers 和 dataGridViewOrders)显示从AutoLot数据库中Inventory、Customers和Orders这3个表取得的数据。另外，还有一个名为btnUpdateDatabase的Button通过数据适配器对象把所有的修改结果返回到数据库处理。

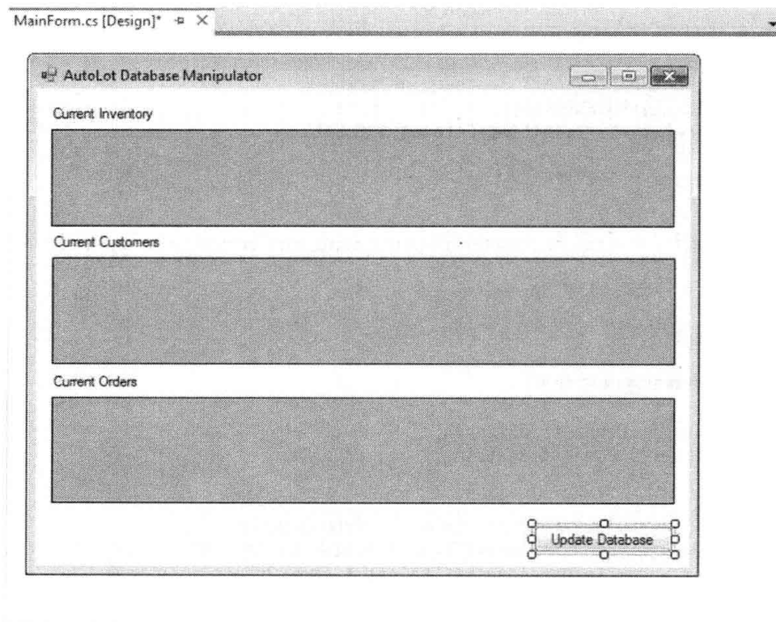


图22-11 初始UI显示从AutoLot数据库的每个表中获得的数据

22.9.1 建立数据适配器

为了简单起见，MainForm还是使用命令构建器对象来为3个SqlDataAdapter（每个表一个）自动生成SQL命令。这里是Form继承的类型的最初实现（不要忘记导入System.Data.Sqlclient命令空间）：

```
public partial class MainForm : Form
{
  // 窗体级别的DataSet
  private DataSet autoLotDS = new DataSet("AutoLot");

  // 使用命令构建器来简化数据适配器的配置
  private SqlCommandBuilder sqlCBInventory;
  private SqlCommandBuilder sqlCBCustomers;
  private SqlCommandBuilder sqlCBOrders;

  // 我们的数据适配器（对于每个表）
  private SqlDataAdapter invTableAdapter;
  private SqlDataAdapter custTableAdapter;
  private SqlDataAdapter ordersTableAdapter;

  // 窗体级别的连接字符串
  private string cnStr = string.Empty;
...
}
```

在窗体的构造函数内我们做一些乏味的操作：建立一些数据相关的成员变量并且填充DataSet。假设这里编写了一个包含正确的连接字符串数据的App.config文件（并引用了System.Configuration.dll和命名空间System.Configuration），例如：

```
<configuration>
  <connectionStrings>
    <add name ="AutoLotSqlProvider"  connectionString =
    "Data Source=(local)\SQLEXPRESS;
      Integrated Security=SSPI;Initial Catalog=AutoLot"
    />
  </connectionStrings>
</configuration>
```

同样注意，这里使用了一个私有的辅助方法BuildTableRelationship()，代码如下：

```
public MainForm()
{
  InitializeComponent();

  // 从*.config文件中提取连接字符串
  cnStr =
    ConfigurationManager.ConnectionStrings[
    "AutoLotSqlProvider"].ConnectionString;

  // 建立数据适配器
  invTableAdapter = new SqlDataAdapter("Select * from Inventory", cnStr);
  custTableAdapter = new SqlDataAdapter("Select * from Customers", cnStr);
  ordersTableAdapter = new SqlDataAdapter("Select * from Orders", cnStr);

  // 自动生成命令
  sqlCBInventory = new SqlCommandBuilder(invTableAdapter);
  sqlCBOrders = new SqlCommandBuilder(ordersTableAdapter);
```

```
    sqlCBCustomers = new SqlCommandBuilder(custTableAdapter);

    // 在DS中填充表
    invTableAdapter.Fill(autoLotDS, "Inventory");
    custTableAdapter.Fill(autoLotDS, "Customers");
    ordersTableAdapter.Fill(autoLotDS, "Orders");

    // 为表间建立关系
    BuildTableRelationship();

    // 绑定到网格
    dataGridViewInventory.DataSource = autoLotDS.Tables["Inventory"];
    dataGridViewCustomers.DataSource = autoLotDS.Tables["Customers"];
    dataGridViewOrders.DataSource = autoLotDS.Tables["Orders"];
  }
```


22.9.2 建立表间关系

BuildTableRelationship()辅助方法将两个DataRelation对象添加到autoLotDs对象。第21章说过AutoLot数据库表示一组父/子关系，下面是具体实现：

```
private void BuildTableRelationship()
{
  // 建立CustomerOrder数据关系对象
  DataRelation dr = new DataRelation("CustomerOrder",
    autoLotDS.Tables["Customers"].Columns["CustID"],
    autoLotDS.Tables["Orders"].Columns["CustID"]);
  autoLotDS.Relations.Add(dr);

  // 建立InventoryOrder数据关系对象
  dr = new DataRelation("InventoryOrder",
    autoLotDS.Tables["Inventory"].Columns["CarID"],
    autoLotDS.Tables["Orders"].Columns["CarID"]);
  autoLotDS.Relations.Add(dr);
}
```

注意，在创建DataRelation对象的时候，我们使用第一个参数来创建友好的字符串友好名（稍后我们会看到这么做的好处），并且指定用于构建关系本身的键。注意父表（第二个构造函数参数）在子表（第三个构造函数参数）之前指定。

22.9.3 更新Database表

现在已经填充了DataSet对象并且从数据源断开，我们能在本地操作每个DataTable。随意在3个DataGridView上增加、更新或者删除值，当需要提交回数据源处理的时候，单击表单上的Update按钮。Click事件的代码如下：

```
private void btnUpdateDatabase_Click(object sender, EventArgs e)
{
  try
  {
    invTableAdapter.Update(autoLotDS, "Inventory");
    custTableAdapter.Update(autoLotDS, "Customers");
    ordersTableAdapter.Update(autoLotDS, "Orders");
  }
  catch (Exception ex)
```

```
    {
      MessageBox.Show(ex.Message);
    }
  }
```

运行应用程序并执行各个更新。当重新运行应用程序时，我们会发现每一个网格都填充了最新的修改。

22.9.4　在关联表中切换

为了演示DataRelation怎么以编程方式实现表之间的切换，我们扩展UI，新增一个名为btnGet-OrderInfo的Button类型、一个相关的文本框txtCustID和一个描述性标签Label（为了可见，将这些控件放到GroupBox中）。图22-12显示了这个应用程序可能的UI。

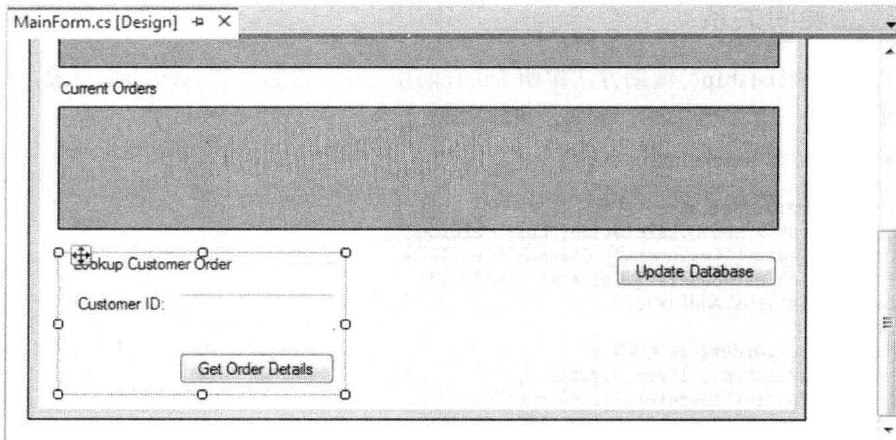


图22-12　更新后的UI允许用户查找客户订单信息

使用这个更新后的UI，最终用户就可以输入客户的ID并且获取客户订单的所有相关信息（name、order、car order等）。它们会被格式化成string类型并最终显示在消息框中。考虑如下新按钮的Click事件处理程序：

```
private void btnGetOrderInfo_Click(object sender, System.EventArgs e)
{
  string strOrderInfo = string.Empty;
  DataRow[] drsCust = null;
  DataRow[] drsOrder = null;

  // 从文本框得到客户的ID
  int custID = int.Parse(this.txtCustID.Text);

  // 根据custID从Customers表得到相应行
  drsCust = autoLotDS.Tables["Customers"].Select(
    string.Format("CustID = {0}", custID));
  strOrderInfo += string.Format("Customer {0}: {1} {2}\n",
    drsCust[0]["CustID"].ToString(),
    drsCust[0]["FirstName"].ToString(),
    drsCust[0]["LastName"].ToString());
```

```
    // 从Customers表切换到Orders表
    drsOrder = drsCust[0].GetChildRows(autoLotDS.Relations["CustomerOrder"]);

    // 为该客户循环所有订单
    foreach (DataRow order in drsOrder)
    {
      strOrderInfo += string.Format("----\nOrder Number:
        {0}\n", order["OrderID"]);

      // 通过订单得到Car的引用
      DataRow[] drsInv = order.GetParentRows(autoLotDS.Relations[
        "InventoryOrder"]);

      // 得到该订单中的（单个）Car信息
      DataRow car = drsInv[0];
      strOrderInfo += string.Format("Make: {0}\n", car["Make"]);
      strOrderInfo += string.Format("Color: {0}\n", car["Color"]);
      strOrderInfo += string.Format("Pet Name: {0}\n", car["PetName"]);
    }

    MessageBox.Show(strOrderInfo, "Order Details");
  }
```


图22-13显示了当指定客户ID为3时可能的输出(由于你的AutoLot数据库表中的数据可能与我的不同，因此结果也会不一样)。

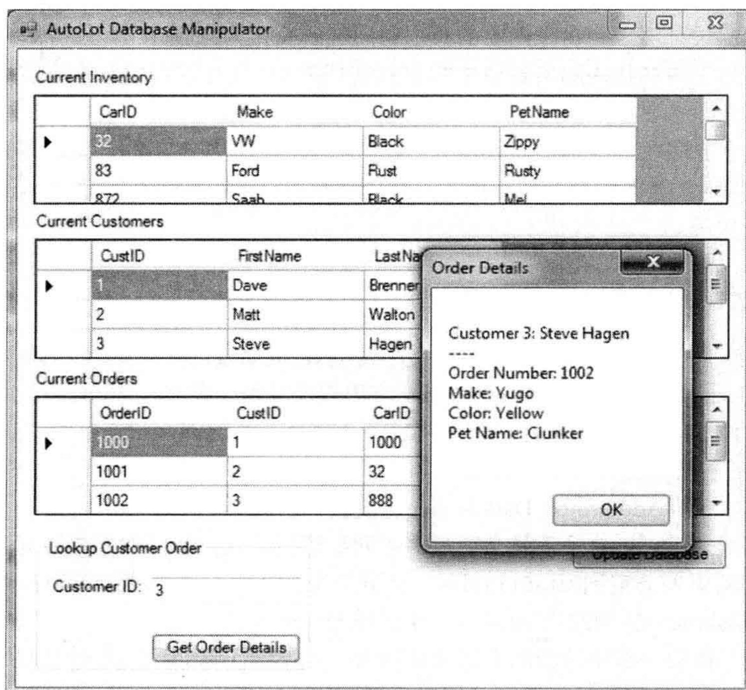


图22-13 使用数据关系导航

希望这最后的例子能让你确信DataSet类有多么有用。既然DataSet是完全和基层数据源断开的，我们就可以使用在内存中的这么一个副本在各个表中切换并做必要的修改、删除和插入操作。操作结束后，就把这些改变提交给数据库去处理。最终的结果是一个可扩展的、健壮的应用程序。

源代码 MultitabledDataSetApp项目的源代码位于Chapter 22子目录下。

22.10 Windows Forms数据库设计器工具

至此，本书中所有的示例都包含了大量的体力活，因为我们都是手动写的这些数据访问逻辑。虽然我们把大量的代码放到.NET类库中（AutoLotDAL.dll）来让书中后面的章节进行重用，在和关系数据库交互之前我们仍然需要手动为我们的数据适配器创建各种对象。

本章接下来将介绍如何使用不同的Windows Forms数据库设计器工具，来为我们生成大量的数据访问代码。

说明 WPF和ASP.NET Web项目包含类似的数据库设计器工具，本章稍后将介绍它们。

使用Windows Forms中DataGridView控件所支持的设计器，是这些集成工具的一种用法。但这种方法的问题是，数据库设计器工具将把所有的数据访问代码直接内嵌到GUI代码库内。理想情况下，应该把所有这些设计器生成的代码隔离到单独的.NET代码库中，这样就可以在多个项目中轻松地复用这些数据库访问逻辑。

不过，由于这个方法在一些小型项目和应用程序原型中还是很有用的，因此我们先来研究如何使用DataGridView控件生成需要的数据访问代码。然后，我们将学习如何在第3版的AutoLot.dll中隔离相同的设计器生成的代码。

22.10.1 可视化设计DataGridView

DataGridView控件有相关的向导，可以为我们生成数据访问代码。首先，创建一个叫DataGridViewDataDesigner的全新Windows Forms应用程序项目。使用Solution Explorer重命名表单为MainForm.cs，然后添加DataGridView控件的实例inventoryDataGridView到表单中。在选中DataGridView控件后，在UI控件的右边应该有一个内联编辑器（如果没有，点击控件右上角的小三角图标）。从Choose Data Source下拉框中选择Add Project Data Source链接，如图22-14所示。

数据源配置向导启动了。这个工具会通过一系列步骤引导我们选择和配置数据源，然后就会使用自定义的数据适配器类型绑定到DataGridView。向导的第一步只是询问我们希望和哪种类型的数据源进行交互。选择Database（如图22-15所示），然后单击Next按钮。

第二步（可能会因第一步中选择的不同而稍微有一点不同）询问我们是希望使用Dataset数据库模型还是Entity数据模型。确保选择Dataset数据库模型（如图22-16所示），因为你还没有学习Entity框架（下一章会详细介绍）。

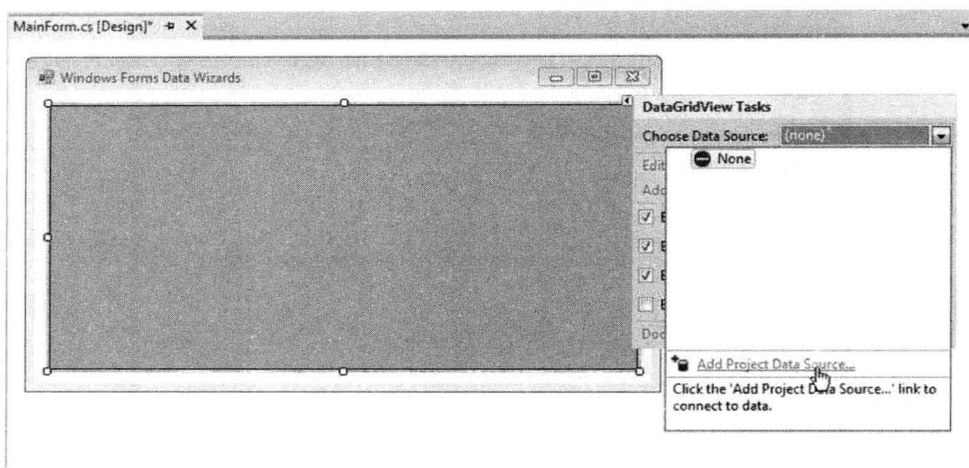


图22-14 DataGridView编辑器

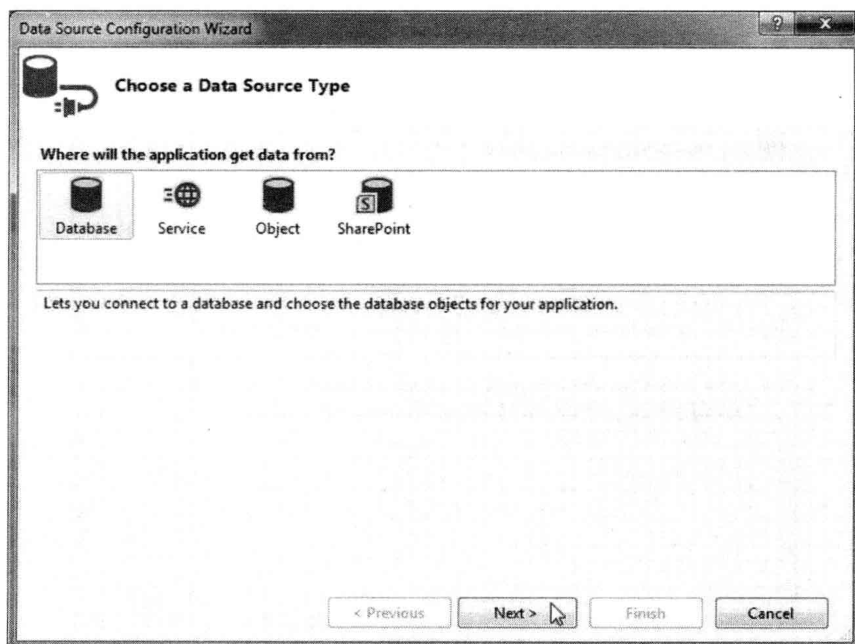


图22-15 选择数据源类型

第三步允许我们配置数据库连接。如果数据库现在已经加到了服务器资源管理器中，我们就可以在下拉列表中发现它自动列出了。如果不是这样的话（或如果你需要连接一个之前没有加到服务器资源管理器中的数据库），单击New Connection按钮。图22-17显示了选择AutoLot本地实例后的结果。

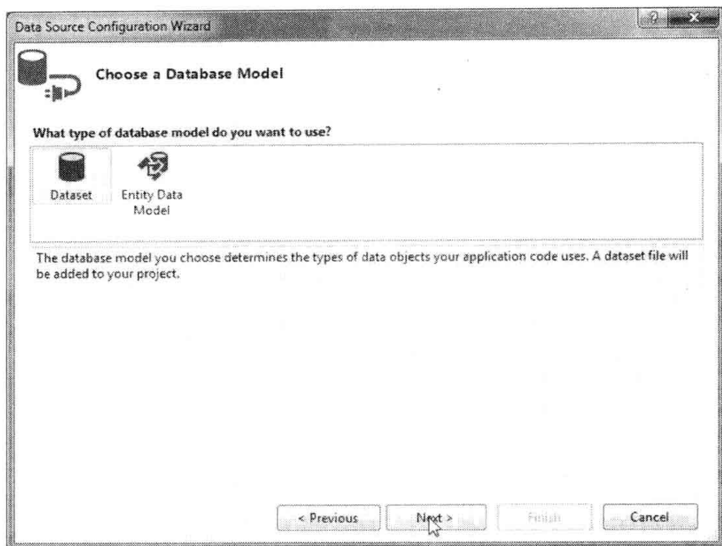


图22-16　选择数据库模型

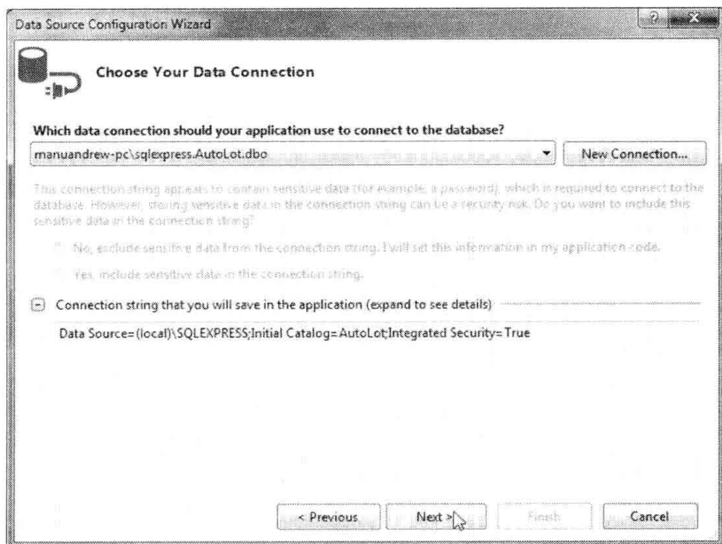


图22-17　选择数据库

在向导的下一步，你将被询问是否希望将连接字符串保存到应用程序配置文件。这里没有给出截屏，你应该选择保存连接字符串，然后点击Next按钮。

最后一步就是选择将由自动生成的DataSet和相关数据适配器处理的数据库对象。虽然我们可以选择AutoLost数据库的所有数据对象，但在这里我们只关心Inventory表。因此，把DataSet的建议名修改为InventoryDataSet（如图22-18所示），检查Inventory表，然后单击Finish按钮。

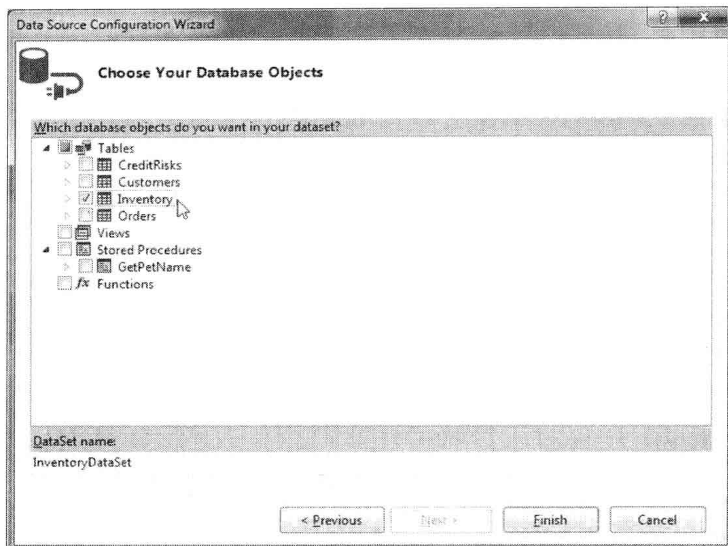


图22-18　选择Inventory表

完成之后，我们就会注意到可视化设计器很多地方都更新了。最值得注意的就是DataGridView显示了Inventory表的构架，如列头所显示的那样。同样，在表单设计器的底部（在组件托盘区域），我们会看到3个组件：DataSet组件、BindingSource组件和TableAdapter组件（如图22-19所示）。

至此，我们就可以运行应用程序并且可以看到，网格已经填充了Inventory表的记录，当然，这并不神奇。IDE已经为我们编写了大量代码并建立了网格控件。下面让我们深入探讨这些自动生成的代码。

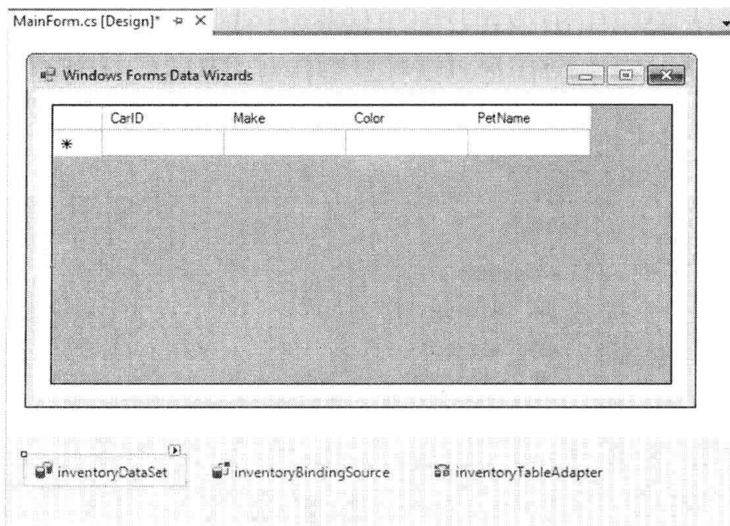


图22-19　运行了数据源配置向导之后的Windows Forms项目

22.10.2　生成的App.config文件

如果研究解决方案资源管理器，就会发现我们的项目现在包含了App.config文件，其中包括一个名字有点奇怪的<connectionStrings>元素，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
  <connectionStrings>
    <add name="DataGridViewDataDesigner.Properties.Settings.AutoLotConnectionString"
      connectionString=
      "Data Source=(local)\SQLEXPRESS;
      Initial Catalog=AutoLot;Integrated Security=True"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

自动生成的数据适配器对象（稍后会详细介绍）使用DataGridViewDataDesigner.Properties.Settings.AutoLotConnection string这个值。

22.10.3　强类型的DataSet

除了配置文件，向导工具还生成了强类型的DataSet。它扩展了DataSet，公开了一些成员，这些成员允许你使用更为直观的对象模型与数据库进行交互。例如，强类型的DataSet对象包含直接映射到数据库表名的属性。因此，可以使用Inventory属性直接获取行和列，而不必使用Tables属性访问表的集合了。首先，通过选择Solution Explorer中的项目图标并且单击View Class Diagram按钮来向我们的项目插入一个新的类图文件。注意，向导根据我们的输入新建了一个InventoryDataSet类。这个类定义了很多成员，其中最重要的就是一个叫Inventory的属性（如图22-20所示）。

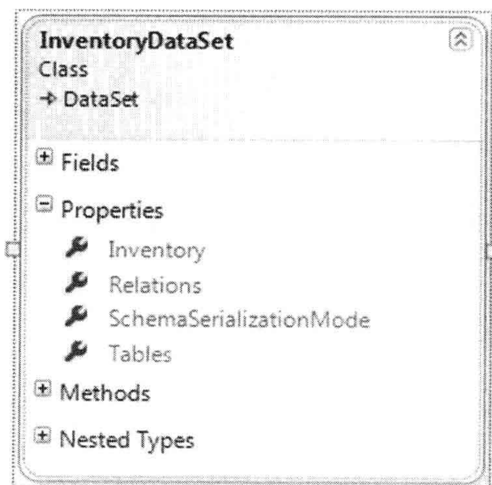


图22-20　数据源配置向导创建了一个强类型的DataSet

如果我们在Solution Explorer中双击InventoryDataSet.xsd文件，就会加载Visual Studio Dataset设计器（稍后会说这个设计器的详细内容）。如果右击设计器的任何地方并且选择View Code选项，就会发现如下所示的空的分部类定义：

```
public partial class InventoryDataSet {
}
```

如果需要，可以在这个分部类中添加自定义成员。但是真正的行为发生在设计器维护的InventoryDataSet.Designer.cs文件中。在Solution Explorer中打开该文件，会发现InventoryDataSet扩展了DataSet父类。考虑如下加上注释的分部代码：

```
// 这是设计器生成的所有代码
public partial class InventoryDataSet : global::System.Data.DataSet
{
  // InventoryDataTable类型的成员变量
  private InventoryDataTable tableInventory;

  // 每一个构造函数调用了一个叫做InitClass()的辅助方法
  public InventoryDataSet()
  {
  ...
    this.InitClass();
  ...
  }

  // InitClass()准备了DataSet并且把InventoryDataTable添加到了Tables集合中
  private void InitClass()
  {
    this.DataSetName = "InventoryDataSet";
    this.Prefix = "";
    this.Namespace = "http://tempuri.org/InventoryDataSet.xsd";
    this.EnforceConstraints = true;
    this.SchemaSerializationMode =
      global::System.Data.SchemaSerializationMode.IncludeSchema;
    this.tableInventory = new InventoryDataTable();
    base.Tables.Add(this.tableInventory);
  }

  // 只读的Inventory属性返回了InventoryDataTable成员变量
  public InventoryDataTable Inventory
  {
    get { return this.tableInventory; }
  }
}
```

注意，强类型的DataSet包含强类型的DataTable成员变量，其类名为InventoryDataTable。强类型DataSet类的构造函数调用一个私有的初始化方法InitClass()，会将这个强类型DataTable的实例添加到DataSet的Tables集合中。最后，注意Inventory属性的实现返回InventoryDataTable成员变量。

22.10.4 强类型的DataTable

现在返回类图文件，打开InventoryDataSet图标下的Nested Types节点，可以发现强类型的DataTable类InventoryDataTable和强类型的DataRow类InventoryRow。

InventoryDataTable类（和我们刚才研究的强类型DataSet中成员变量的类型一样）根据物理

Inventory表的列名定义了一组属性（CarIDColumn、ColorColumn、MakeColumn和PetNameColumn）、自定义索引器以及获取当前记录数的Count属性。

最有趣的是，这个强类型的DataTable类定义了一组方法，允许我们使用强类型成员（手动导航Rows和Columns索引器的更佳方法）在表中插入、定位和删除行。

例如，AddInventoryRow()可以向内存中的表添加新的记录行，FindByCarID()可以根据表的主键进行查找，RemoveInventoryRow()可以从强类型的表中移除一行（如图22-21所示）。

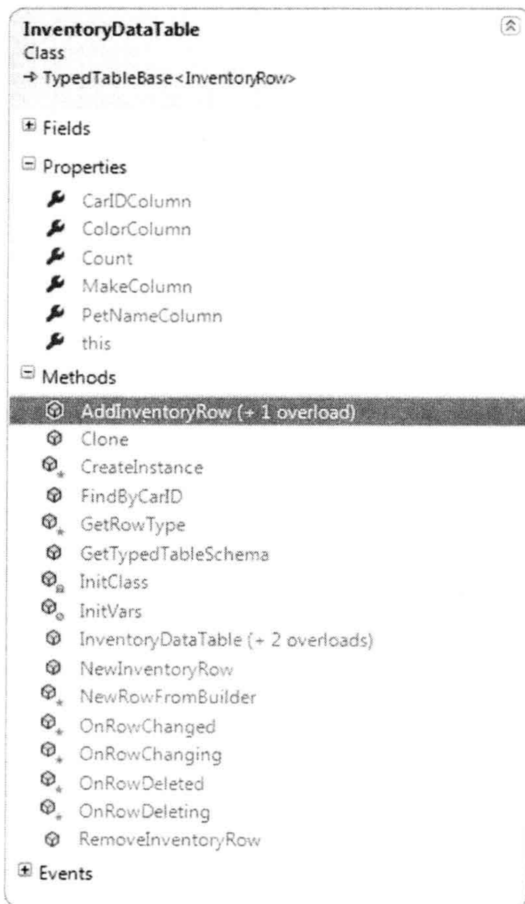


图22-21　强类型DataSet嵌套在强类型的DataTable中

22.10.5　强类型的DataRow

同样内嵌在强类型DataSet中的强类型DataRow类扩展了DataRow类，并公开了一些直接映射到Inventory表架构的属性。同样，数据设计器工具生成了一个方法（IsPetNameNull()），检查某列是否有值（如图22-22所示）。

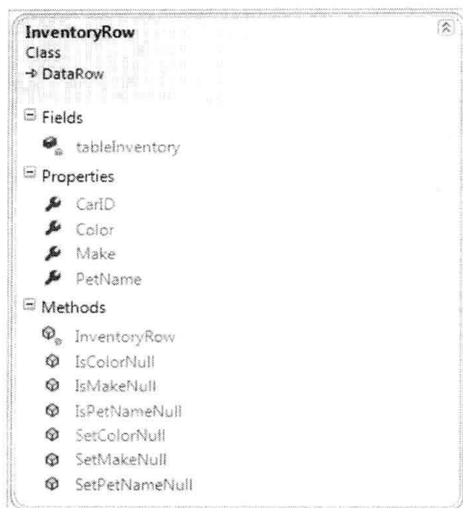


图22-22 强类型DataRow

22.10.6 强类型的数据适配器

使用数据源配置向导的一个非常大的优势就是能让我们的断开连接类型有很多强类型，因为手工创建这些类是很无聊的（但也是完全有可能的）。这个向导还生成了自定义的数据适配器对象，可以以强类型的方式填充和更新InventoryDataSet和InventoryDataTable对象。在可视化类设计器中找到InventoryTableAdapter，并查看生成的成员，如图22-23所示。

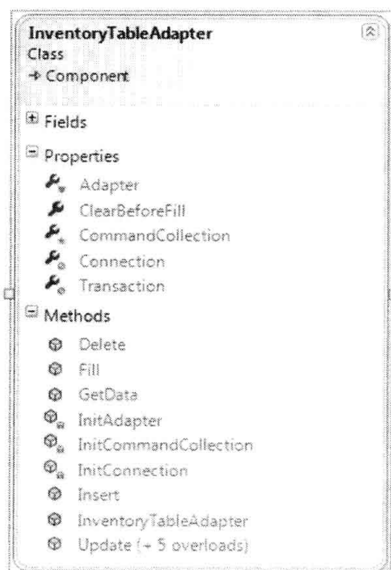


图22-23 自定义的数据适配器操作强类型的DataSet和DataTable

自动生成的InventoryTableAdapter类型维护SqlCommand对象（可以使用CommandCollection属性来访问它）的集合，每一个都填充了一组SqlParameter对象。此外，自定义的数据适配器提供了一组属性来提取底层的连接、事务以及数据适配器对象，还有一组属性用来获取表示每一个命令类型的数组。

22.10.7 完成Windows Forms应用程序

如果研究表单派生类型的Load事件处理程序（即如果查看MainForm.cs的代码并且找到MainFormLoad()方法），就会发现自定义数据适配器的Fill()方法在启动的时候就被调用，并且传入由自定义DataSet维护的自定义DataTable：

```
private void MainForm_Load(object sender, EventArgs e)
{
  this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);
}
```

我们可以使用相同的自定义数据适配器对象来更新网格。使用一个Button控件（叫做btnUpdateInventory）更新表单的UI。接着处理Click事件并且在事件处理程序中编写如下代码：

```
private void btnUpdateInventory_Click(object sender, EventArgs e)
{
  try
  {
    // 这会把Inventory表中的任何修改发回数据库进行处理
    this.inventoryTableAdapter.Update(this.inventoryDataSet.Inventory);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }

  // 为网格获取新的数据
  this.inventoryTableAdapter.Fill(this.inventoryDataSet.Inventory);
}
```

再一次运行应用程序，增加、删除或更新显示在网格中的记录，并且单击Update按钮。如果我们再次运行程序，就会发现我们的改动生效并且被处理了。

该示例告诉我们DataGridView控件设计器是多么有帮助。它带来的强类型数据为我们生成了大量必要的数据库逻辑。一个明显的问题是，这些代码与使用它们的窗口紧密相连。理想情况下，这些代码应该属于AutoLotDAL.dll程序集（或其他数据访问库）。但你可能想知道如何在Class Libarary项目中使用DataGridView的向导所生成的代码，因为默认情况下没有窗体设计器。

源代码 DataGridViewDataDesigner项目的源代码位于Chapter 22子目录下。

22.11 将强类型的数据库代码隔离到类库中

幸好我们可以从任何类型的项目（基于UI或其他）中启动Visual Studio的数据设计工具，无须在项目之间复制和粘贴大量的代码。向AutoLotDAL.dll添加更多功能即可看到这一点。

将本章前面创建的AutoLotDAL(Version Two)项目所在的文件夹整个复制到硬盘的新位置，并将文件夹重命名为AutoLotDAL(Version Three)。接下来，单击Visual Studio的File→Open Project/Solution...菜单选项，打开AutoLotDAL(Version Three)目录下的AutoLotDAL.sln文件。

通过Project→Add New Item菜单项在项目中插入一个新的DataSet类（叫做AutoLotDataSet.xsd）（要快速找到DataSet项目类型，可以选择New Item对话框的Data节，如图22-24所示）。

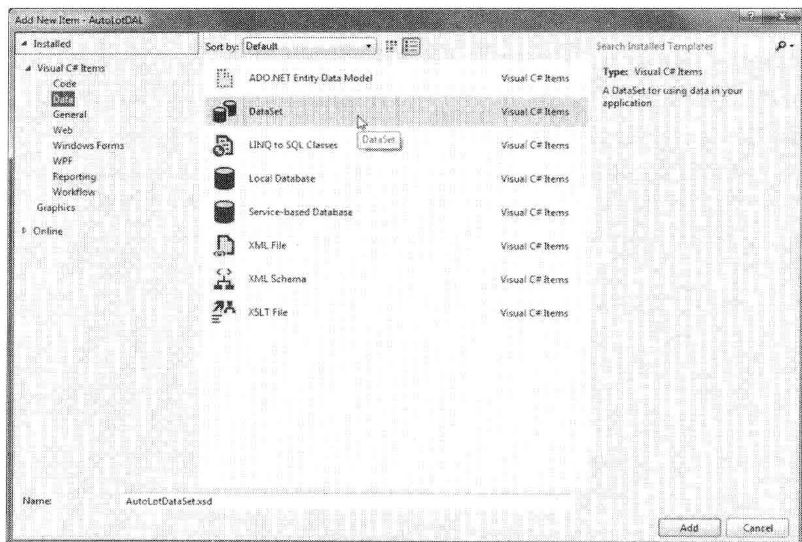


图22-24 插入新的强类型的DataSet

这就会打开一个空的DataSet设计器界面。此时，使用服务器资源管理器连接某个数据库（你应该已经有AutoLot的连接了），并且将每一个希望生成的数据库对象拖到界面上。从图22-25中，可以看到AutoLot的每一个自定义方面，并且它们的关系被自动实现（这里就不拖CreditRisk表了）。

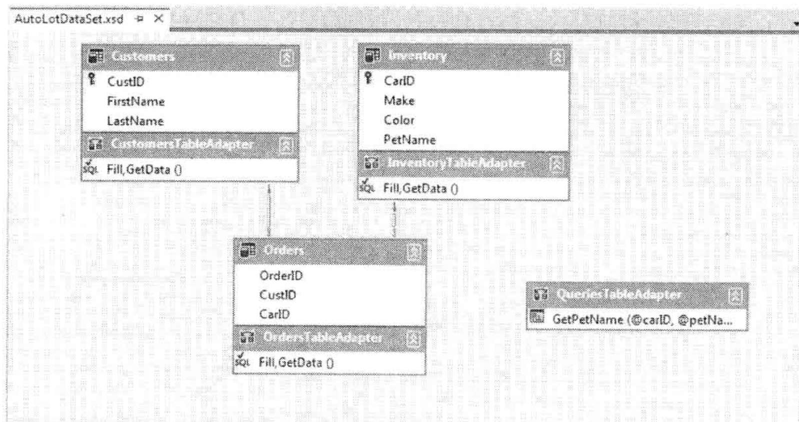


图22-25 自定义的强类型类型，这次在类库项目中

22.11.1　查看生成的代码

DataSet设计器所创建的代码，与前面的Windows Forms示例中DataGridView向导所创建的代码完全相同。只不过这次我们生成了Inventory、Customers和Orders表，以及GetPetName存储过程，因此有了更多的生成类。基本上，每个拖曳到设计器界面上的数据库表都将得到强类型的DataSet，以及其内部的DataTable、DataRow和数据适配器类。

强类型的DataSet、DataTable、DataRow类将位于项目（AutoLotDAL）的根命名空间中。自定义的表适配器将位于内嵌的命名空间中。你可以打开Visual Studio View菜单下的Class View工具，非常容易地查看所有生成的类型（如图22-26所示）。

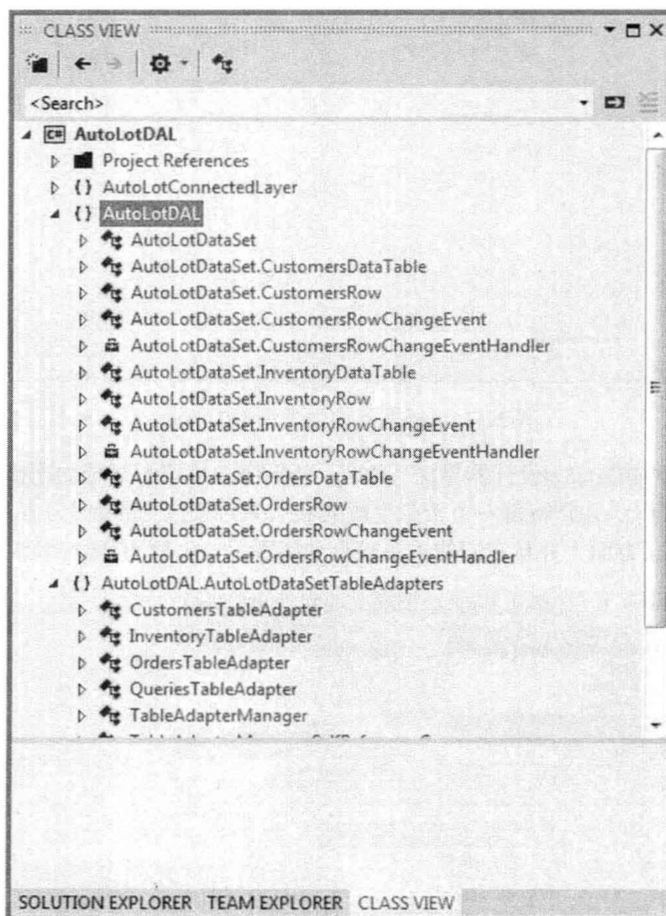


图22-26　AutoLot数据库自动生成的强类型数据

为完备起见，你可能希望使用Visual Studio的Properties编辑器（详见第14章），将AutoLotDAL.dll的版本设置为3.0.0.0。

源代码　AutoLotDAL (Version Three)项目的源代码位于Chapter 22子目录下。

22.11.2　用生成的代码选择数据

此时，我们可以在需要与AutoLot数据库通信的.NET应用程序中使用这些强类型数据。为了确保你已经理解了基本机制，创建一个控制台应用程序StronglyTypedDataSetConsoleClient。然后，添加对于最新版的AutoLotDAL.dll的引用，并在初始的C#代码文件中引入AutoLotDAL和AutoLotDAL.AutoLotDataSetTableAdapters命名空间。

下面的Main()方法使用InventoryTableAdapter对象获取Inventory表中的所有数据。注意，你没有必要指定连接字符串，因为它已经成为强类型对象模型的一部分了。填充数据表后，使用辅助方法PrintInventory()打印结果。注意，对于强类型的DataTable，你完全可以像使用Rows和Columns集合操作“普通”的DataTable那样进行操作：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Strongly Typed DataSets *****\n");

    AutoLotDataSet.InventoryDataTable table =
      new AutoLotDataSet.InventoryDataTable();

    InventoryTableAdapter dAdapt = new InventoryTableAdapter();
    dAdapt.Fill(table);

    PrintInventory(table);
    Console.ReadLine();
  }

  static void PrintInventory(AutoLotDataSet.InventoryDataTable dt)
  {
    // 打印列名
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Console.Write(dt.Columns[curCol].ColumnName + "\t");
    }
    Console.WriteLine("\n----------------------------------");

    // 打印数据
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
      for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
      }
      Console.WriteLine();
    }
  }
}
```

22.11.3　用生成的代码插入数据

现在假设你要用这个强类型的对象模型插入新的记录。下面的辅助方法向当前的InventoryDataTable中添加两个新行，然后使用数据适配器更新数据库。第一行通过配置强类型DataRow手工添加，第二行通过传递所需的列数据添加，该列数据用来在后台自动创建DataRow：

```
public static void AddRecords(AutoLotDataSet.InventoryDataTable tb,
  InventoryTableAdapter dAdapt)
{
try
{
  // 从表中获取新的强类型的行
  AutoLotDataSet.InventoryRow newRow = tb.NewInventoryRow();

  // 使用一些示例数据填充该行
  newRow.CarID = 999;
  newRow.Color = "Purple";
  newRow.Make = "BMW";
  newRow.PetName = "Saku";

  // 插入新行
  tb.AddInventoryRow(newRow);

  // 使用重载的Add方法添加另一个行
  tb.AddInventoryRow(888, "Yugo", "Green", "Zippy");

  // 更新数据库
  dAdapt.Update(tb);
}
Catch(Exception ex)
{
    Console. WriteLine(ex.Message);
}
       }
```

Main()方法调用该方法，用这些新的记录更新数据库：

```
static void Main(string[] args)
{
...
  // 添加行，更新并再次打印
  AddRecords(table, dAdapt);
  table.Clear();
  dAdapt.Fill(table);
  PrintInventory(table);
  Console.ReadLine();
}
```

22.11.4　用生成的代码删除数据

使用强类型的对象模型删除数据同样很简单。强类型的DataTable中包含自动生成的FindBy*XXXX*()方法（其中*XXXX*为主键列的名称）根据主键返回正确的（强类型的）DataRow。下面的辅助方法将删除刚刚创建的两行：

```
private static void RemoveRecords(AutoLotDataSet.InventoryDataTable tb,
  InventoryTableAdapter dAdapt)
```

```
{
  try
  {
    AutoLotDataSet.InventoryRow rowToDelete = tb.FindByCarID(999);
    dAdapt.Delete(rowToDelete.CarID, rowToDelete.Make,
                  rowToDelete.Color, rowToDelete.PetName);
    rowToDelete = tb.FindByCarID(888);
    dAdapt.Delete(rowToDelete.CarID, rowToDelete.Make,
                  rowToDelete.Color, rowToDelete.PetName);
  }
  Catch(Exception ex)
  {
    Console. WriteLine(ex.Message);
  }
}
```

如果在Main()方法中调用该方法并重新打印该表，那么将不再显示两条测试记录。

说明 如果运行该应用程序两次并且每次都调用AddRecord()方法，将得到违反约束错误（VOILATION CONSTRAINT ERROR），因为AddRecord()方法每次都试图插入同样的CarID主键值（所以我们才将数据访问逻辑放到try/catch里）。要想让该示例更灵活，应该使用Console类，让客户输入数据。

22.11.5 用生成的代码调用存储过程

让我们来看看另一个使用强类型对象模型的示例。在该示例中，我们创建一个调用GetPetName存储过程的方法。当AutoLot数据库的数据适配器被创建时，有一个特殊的类QueriesTableAdapter。顾名思义，它可以用来在关系数据库中调用存储过程。在Main()中调用下面的辅助方法将显示指定的汽车名称：

```
public static void CallStoredProc()
{
  try
  {
    QueriesTableAdapter q = new QueriesTableAdapter();
    Console.Write("Enter ID of car to look up: ");
    string carID = Console.ReadLine();
    string carName = "";
    q.GetPetName(int.Parse(carID), ref carName);
    Console.WriteLine("CarID {0} has the name of {1}", carID, carName);
  }
  Catch(Exception ex)
  {
      Console. WriteLine(ex.Message);
  }
}
```

现在，我们知道了如何使用强类型的数据库类型，以及如何把它们包装到单独的类库中。关于这个对象模型，还有很多方面没有介绍，但如果感兴趣你可以自行研究。在本章最后，你将学习如何对ADO.NET DataSet对象使用LINQ查询。

源代码 StronglyTypedDataSetConsoleClient项目的源代码位于Chapter 22子目录下。

22.12　LINQ to DataSet

我们在本章中学习了3种不同的操作DataSet数据的方式：

- ❑ 使用Tables、Rows和Columns集合；
- ❑ 使用数据表读取器；
- ❑ 使用强类型数据类。

在使用DataSet和DataTable等类型的索引器时，可以直接以弱类型的方式与数据交互。记住这种方法要求我们将数据视为表格，例如：

```
static void PrintDataWithIndxers(DataTable dt)
{
  // 打印DataTable
  for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
  {
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
    }
    Console.WriteLine();
  }
}
```

DataTable类型还提供了CreateDataReader()方法，将DataSet中的数据视为可顺序处理的线性的行。这使得你可以在非连接的DataSet中使用连接的数据读取器编程模型：

```
static void PrintDataWithDataTableReader(DataTable dt)
{
  // 获取DataTableReader类型
  DataTableReader dtReader = dt.CreateDataReader();
  while (dtReader.Read())
  {
    for (int i = 0; i < dtReader.FieldCount; i++)
    {
      Console.Write("{0}\t", dtReader.GetValue(i));
    }
    Console.WriteLine();
  }
  dtReader.Close();
}
```

最后，我们可以使用强类型的DataSet来生成代码库，该代码库允许你使用映射到关系数据库各个列名的属性与对象中的数据进行交互。使用强类型的对象，可以编写下面这样的代码：

```
static void AddRowWithTypedDataSet()
{
  InventoryTableAdapter invDA = new InventoryTableAdapter();
  AutoLotDataSet.InventoryDataTable inv = invDA.GetData();
  inv.AddInventoryRow(999, "Ford", "Yellow", "Sal");
  invDA.Update(inv);
}
```

尽管每种方法都有其用武之地，但LINQ to DataSet API提供了一种使用LINQ查询表达式来操纵DataSetAPI的选择。

说明 我们使用LINQ to DataSet API只对数据适配器返回的DataSet对象执行LINQ查询，而不是对数据库引擎本身直接应用LINQ查询。第23章将介绍LINQ to Entity和ADO.NET Entity Framework，它可以以LINQ查询来表示SQL查询。

ADO.NET DataSet（及其相关的类型，如DataTable和DataView）并不具备成为LINQ查询目标的基础结构。例如，下面的方法（使用了AutoLotDisconnectedLayer命名空间中的类型）将导致编译时错误：

```
static void LinqOverDataTable()
{
  // 获取数据的DataTable
  InventoryDALDisLayer dal = new InventoryDALDisLayer(
    @"Data Source=(local)\SQLEXPRESS;" +
     "Initial Catalog=AutoLot;Integrated Security=True");
  DataTable data = dal.GetAllInventory();

  // 对DataSet应用LINQ查询
  var moreData = from c in data where (int)c["CarID"] > 5 select c;
}
```


编译LinqOverDataTable()方法，编译器将通知你DataTable类型需要提供查询模式的实现。这与对未实现IEnumerable<T>的对象执行LINQ查询的过程一样，ADO.NET对象必须转换为兼容的类型。要理解这些，需要先研究System.Data.DataSetExtensions.dll的类型。

22.12.1 DataSet Extensions库的作用

Visual Studio项目默认引用了System.Data.DataSetExtensions.dll程序集，它用少量的新类型扩展了System.Data命名空间（如图22-27所示）。

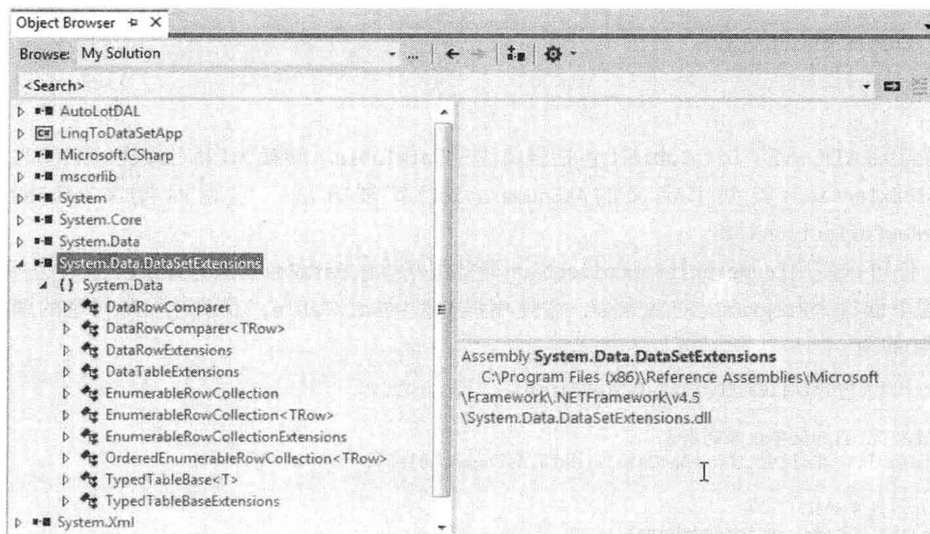


图22-27 System.Data.DataSetExtensions.dll程序集

最常用的两个类为DataTableExtensions和DataRowExtensions。它们使用一些扩展方法扩展了DataTable和DataRow的功能（见第12章）。另一个关键的类是TypedTableBaseExtensions，它也定义了一些扩展方法，可以用来对强类型DataSet对象内部的DataTable执行LINQ查询。System.Data.DataSetExtensions.dll程序集中的其他成员都是纯粹的基础结构，不要在代码库中直接使用它们。

22.12.2　获取与LINQ兼容的DataTable

现在让我们来看看如何使用这个DataSet扩展。假设我们有一个新的C#控制台应用程序，名为LinqToDataSetApp。添加最新版（3.0.0.0）的AutoLotDAL.dll程序集，用以下逻辑更新初始的代码文件：

```
using System;
...
// 强类型数据容器所在的位置
using AutoLotDAL;

// 强类型数据适配器所在的位置
using AutoLotDAL.AutoLotDataSetTableAdapters;

namespace LinqToDataSetApp
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** LINQ over DataSet *****\n");

// 获取包含AutoLot数据库中当前Inventory的强类型DataTable
      AutoLotDataSet dal = new AutoLotDataSet();
      InventoryTableAdapter da = new InventoryTableAdapter();
      AutoLotDataSet.InventoryDataTable data = da.GetData();

      // 在下面调用那些方法

      Console.ReadLine();
    }
  }
}
```

当你希望将ADO.NET DataTable（包括强类型的DataTable）转换为LINQ兼容的对象时，必须调用DataTableExtensions类型中定义的AsEnumerable()扩展方法。它返回包含DataRows集合的EnumerableRowCollection对象。

然后，你可以使用EnumerableRowCollection类型用基本的DataRow语法（如索引器语法）来操作每一行。考虑下面这个Program类的新方法，它使用强类型的DataTable，获取数据可枚举的副本，并打印每个CarID的值：

```
static void PrintAllCarIDs(DataTable data)
{
  // 获取DataTable可枚举的副本
  EnumerableRowCollection enumData = data.AsEnumerable();

  // 打印汽车的ID
  foreach (DataRow r in enumData)
    Console.WriteLine("Car ID = {0}", r["CarID"]);
}
```

这里并没有使用LINQ查询。但关键是这里的enumData对象可以成为LINQ查询表达式的目标。再次注意，EnumerableRowCollection包含DataRow对象的集合，因为我们对每个子对象都使用了类型索引器，来打印CarID列的值。

在大多数情况下，我们没有必要声明一个EnumerableRowCollection类型的变量来保存AsEnumerable()的返回值，而是可以在查询表达式内部调用该方法。以下是Program类中一个更有趣的方法，它从所有汽车颜色为红色的（如果Inventory表中没有红色的汽车，则可以更改该LINQ查询）DataTable中获取CarID + Makes的投影：

```
static void ShowRedCars(DataTable data)
{
  // 对Color = Red的行，投影包含ID/颜色的结果集
  var cars = from car in data.AsEnumerable()
             where
               (string)car["Color"] == "Red"
             select new
             {
               ID = (int)car["CarID"],
               Make = (string)car["Make"]
             };
  Console.WriteLine("Here are the red cars we have in stock:");
  foreach (var item in cars)
  {
    Console.WriteLine("-> CarID = {0} is {1}", item.ID, item.Make);
  }
}
```


22.12.3 DataRowExtensions.Field<T>()扩展方法的作用

当前LINQ查询表达式一个不尽如人意的地方是，我们在获取结果集时使用了大量的转换操作和DataRow索引器，如果试图转换不兼容的数据类型，将会导致运行时异常。要在查询中注入一些强类型，可以使用DataRow类型的Field<T>()扩展方法。这么做可以增强查询的类型安全，因为数据类型的兼容性将在编译时进行检查。考虑如下的更新：

```
var cars = from car in data.AsEnumerable()
  where
    car.Field<string>("Color") == "Red"
  select new
  {
    ID = car.Field<int>("CarID"),
    Make = car.Field<string>("Make")
  };
```

在本例中，我们调用Field<T>()，指定表示列实际数据类型的类型参数。该方法的参数为列名本身。由于作了额外的编译时检查，所以处理EnumerableRowCollection的最佳实践是使用Field<T>()（而不是DataRow索引器）。

除了调用AsEnumerable()方法，整个LINQ查询的格式与第13章介绍的基本相同。因此，没有必要在此重复不同LINQ操作的细节了。如果希望看到更多的示例，可以查看.NET Framework 4.5 SDK文档中的“LINQ to DataSet Examples”话题。

22.12.4　从LINQ查询中生成新的DataTable

如果不使用投影，也可以基于LINQ查询的结果生成新的DataTable数据。当结果集的实际类型可以表示为IEnumerable<T>时，可以调用CopyToDataTable()扩展方法，如下例所示：

```
static void BuildDataTableFromQuery(DataTable data)
{
  var cars = from car in data.AsEnumerable()
             where
               car.Field<int>("CarID") > 5
             select car;

  // 使用该结果集来构建新的DataTable
  DataTable newTable = cars.CopyToDataTable();
  // 打印DataTable
  for (int curRow = 0; curRow < newTable.Rows.Count; curRow++)
  {
    for (int curCol = 0; curCol < newTable.Columns.Count; curCol++)
    {
      Console.Write(newTable.Rows[curRow][curCol].ToString().Trim() + "\t");
    }
    Console.WriteLine();
  }
}
```

说明　还可以使用AsDataView<T>()扩展方法将LINQ查询转换为DataView类型。

当你将LINQ查询的结果作为数据绑定操作的源时，你会发现这项技术非常有帮助。Windows Forms（以及ASP.NET或WPF网络控件）中的DataGridView支持DataSource属性。你可以像下面这样绑定LINQ结果：

```
// 假设myDataGrid为基于GUI的网格对象
myDataGrid.DataSource = (from car in data.AsEnumerable()
            where
              car.Field<int>("CarID") > 5
            select car).CopyToDataTable();
```

现在完成了对于ADO.NET非连接层的介绍。使用这些API，可以从关系数据库中获取、处理并返回数据，而数据库连接的打开时间则尽可能短。

源代码　LinqOverDataSetApp示例的源代码位于Chapter 22子目录下。

22.13　小结

本章深入探讨了ADO.NET断开连接层的细节。我们已经看到了，断开连接层的核心是DataSet。这个类型是许多表、可选关系、约束以及表达式在内存中的表现。在本地数据表之间构建关系的魅力在于，从远程数据库断开的情况下我们可以以编程方式在在表之间切换。

在本章中，我们还研究了数据适配器类型的作用。使用这个类型（以及相关的SelectCommand、InsertCommand、UpdateCommand和DeleteCommand属性），适配器就可以同步原始数据库源和DataSet。同样，我们学习了如何利用由Visual Studio的DataSet设计器工具生成的强类型对象，通过强大的手动方式来使用DataSet的对象模型。

最后介绍了一种LINQ技术LINQ to DataSet。它能获得DataSet可查询的副本，接收格式化的LINQ查询。

第23章 ADO.NET之三：Entity Framework

前两章介绍了ADO.NET编程模型的底层机制——连接层和断开连接层。从最初发布的平台开始，.NET程序员就是用这些方法（以一种相对简单的方式）来操作关系型数据。不过，微软在.NET 3.5 Service Pack 1中为ADO.NET API引入了一个全新的组件，称为Entity Framework（简称EF）。

EF的首要目标是使用直接映射到应用程序中业务对象的对象模型与关系型数据库进行交互。例如，它没有将数据视为行和列的集合，而是将其视为强类型对象（称为实体）的集合。这些实体也可用于LINQ，你可以使用第12章所学过的LINQ语法来进行查询。EF运行时引擎会将你的LINQ查询转换为正确的SQL查询。

本章将介绍EF编程模型。你将学习各种基本概念，包括对象服务、实体客户端、LINQ to Entity和Entity SQL。你还将学习到十分重要的*.edmx文件的格式及其在Entity Framework API中的作用。然后还将学习如何使用Visual Studio 或EMD生成工具（edmgen.exe）的命令行来生成*.edmx文件。

阅读完本章之后，你将得到AutoLotDAL.dll的最终版本，并将学习如何在Windows Form桌面应用程序中绑定实体对象。

23.1 Entity Framework 的作用

ADO.NET的连接层和断开连接层提供了一个架构，可以通过连接、命令、数据读取器、数据适配器和DataSet对象来进行查询、插入、更新和删除数据等操作。尽管这是一个十分优秀的架构，但它还是会迫使我们以一种与物理数据库结构紧耦合的方式来处理数据。例如在使用连接层时，我们通常在数据读取器中指定列名来遍历每条记录。而如果使用断开连接层，则将会对DataSet容器中的DataTable对象的行和列的集合进行遍历。

如果在断开连接层中使用强类型的DataSet或数据适配器，那么带来的编程方面的抽象性是很有帮助的。首先，强类型的DataSet使用类的属性来表示表数据。其次，强类型的表适配器支持封装了构造底层SQL语句的方法。如第22章的AddRecords()方法：

```
public static void AddRecords(AutoLotDataSet.InventoryDataTable tb,
                              InventoryTableAdapter dAdapt)
{
  // 从表中得到强类型的新行
  AutoLotDataSet.InventoryRow newRow = tb.NewInventoryRow();
```

```
    // 用示例数据填充行
    newRow.CarID = 999;
    newRow.Color = "Purple";
    newRow.Make = "BMW";
    newRow.PetName = "Saku";

    // 插入新行
    tb.AddInventoryRow(newRow);

    // 使用重载的Add方法添加新行
    tb.AddInventoryRow(888, "Yugo", "Green", "Zippy");

    // 更新数据库
    dAdapt.Update(tb);
  }
```

如果在断开连接层中使用LINQ to DataSet，情况会变得更好。你可以对内存中的数据使用LINQ查询来得到新的记录集，然后再将其映射到单独的对象，如新的DataTable、List<T>、Dictionary<K,V>或数组，如下所示：


```
static void BuildDataTableFromQuery(DataTable data)
{
  var cars = from car in data.AsEnumerable()
             where car.Field<int>("CarID") > 5 select car;

  // 试用该数据集来构建新的DataTable
  DataTable newTable = cars.CopyToDataTable();

  // 操作DataTable
}
```

LINQ to DataSet是非常有用的，但要记住我们使用LINQ查询的目标是得到*数据库中返回的数据*，而不是数据库引擎本身。理想情况下，我们希望构建一个LINQ查询并将其直接发送给数据库引擎进行处理，然后返回强类型的数据（这正是ADO.NET Entity Framework能够做到的）。

不管是使用ADO.NET的连接层还是断开连接层，你都必须十分留意后端数据库的物理结构。你必须了解每个数据表的架构，能够编写复杂的SQL查询与表数据进行交互，等等。这会迫使你编写十分冗长的C#代码，因为C#本身无法直接用于数据库架构。

更糟的是，DBA在构建物理数据库时考虑更多的是数据库的结构，如外键、视图、存储过程。而且出于安全性和可扩展性的角度考虑，数据库会变得更加复杂。这同样会使与数据存储进行交互的C#代码臃肿不堪。

ADO.NET Entity Framework（EF）是一种编程模型，旨在减少数据库结构与面向对象编程结构之间的差异。使用EF，你不用编写SQL代码就能与关系型数据库交互。（如果你愿意的话。）并且，在对强类型类进行LINQ查询时，EF还会在运行时生成正确的SQL语句。

说明 术语LINQ to Entity是指对ADO.NET EF实体对象使用LINQ查询。

相比更新数据库时使用一些SQL查询来查找、更新，再将其发送回数据库进行处理，使用EF可以简单地修改对象的属性，并保存其状态。EF在运行时将自动更新数据库。

对于微软来说，ADO.NET Entity Framework只是数据访问API的另一种途径，其目的并非是取代

连接层和断开连接层。不过当你使用了EF一段时间之后会很快发现，相比原始的SQL查询和行列集合，你会更喜欢这种富对象模型。

但是，在.NET项目中这三种方法都会有不少机会。在某些情况下，EF模型会使代码库显得复杂。例如，在构建一个只与一张数据库表通信的内部应用程序时，你也许会使用连接层来操作一些相关的存储过程。大型应用程序（特别是开发团队熟练掌握LINQ时），使用EF将从中受益。和其他新技术一样，你需要决定如何（以及何时）使用ADO.NET EF来完成手头的工作。

说明　你也许想起了.NET 3.5中引入的一个数据库编程API——LINQ to SQL。该API在概念上（以及编程结构上）与ADO.NET EF接近。虽然LINQ to SQL还没有正式死亡，不过微软官方的说法是希望用户致力于EF，而不是LINQ to SQL。

23.1.1　实体的作用

前面所提到的强类型类称为实体。实体是将物理数据库映射到业务领域的概念模型。这种模型的正式名称为实体数据模型（EDM）。它是一组映射到物理数据库的客户端类，但是这些实体没有必要与数据库架构的命名约定完全一致。你完全可以根据需要调整实体类，EF运行时将把这些唯一的名字映射到正确的数据库架构。

例如，我们使用如图23-1所示的数据库架构在AutoLot数据库中创建简单的Inventory表。

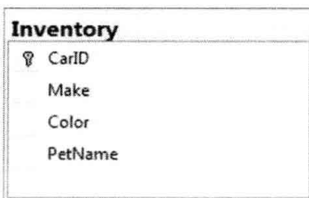


图23-1　AutoLot数据库中Inventory表的结构

如果为AutoLot数据库的Inventory表生成EDM（稍后将介绍如何生成），默认情况下实体的名称为Inventory。但是你也可以将类名该为Car并定义不同名称的属性，这些属性将映射到Inventory表的各个列。这种松耦合意味着可以改变实体使其与业务领域更加接近。图23-2显示了一个实体类。



图23-2　实体Car在客户端对Inventory架构进行了修改

说明 在大多数情况下，客户端实体类的名称都与关系型数据库表名相同。但你总是可以修改它的名称使其与业务环境更相符。

我们稍后会用EF构建完整的示例。但现在，先考虑下面的Program类，它使用Car实体类（和一个与之相关联的AutoLotEntities类）向AutoLot的Inventory表中添加一行新的数据。AutoLotEntites类称为对象下上文，它的作用是与物理数据库进行交互（稍后将学习更详细的内容）：

```
class Program
{
  static void Main(string[] args)
  {
    // 从生成的配置文件中自动读取连接字符串
    using (AutoLotEntities context = new AutoLotEntities())
    {
      // 使用实体向Inventory表中添加一行新的记录
      context.Cars.AddObject(new Car() { AutoIDNumber = 987, CarColor = "Black",
                                        MakeOfCar = "Pinto",
                                        NicknameOfCar = "Pete" });
      context.SaveChanges();
    }
  }
}
```


客户端用什么来代表Inventory表（这里是类Car）并且将其映射到表中的各列，是取决于EF运行时的。你会注意到这里没有任何形式的SQL INSERT语句，你只需要新建一个Car对象并将其添加到上下文对象的Cars属性这个集合中，然后再保存更改。毫无疑问，当使用Visual Studio的Server Explorer查看表中的数据时，会发现多了一行新记录（如图23-3所示）。

Inventory: Query(m...sqlexpress.AutoLot)

	CarID	Make	Color	PetName
	16	Ford	White	FooFoo
	32	VW	Black	Zippy
	83	Ford	Rust	Rusty
	872	Saab	Black	Mel
	888	Yugo	Yellow	Clunker
▶	987	Pinto	Black	Pete
	1000	BMW	Black	Bimmer
	1011	BMW	Green	Hank
	2911	BMW	Pink	Pinky
	12345	Yugo	Green	Zippy
*	*NULL*	*NULL*	*NULL*	*NULL*

6 of 10

图23-3 保存上下文之后的结果

上面的示例没有什么神奇可言。在后台，EF为我们创建一个数据库连接、生成一条正确的SQL语句，等等。为我们处理这些细节是EF的最大优势。现在我们来看看EF是如何实现这些核心服务的。

23.1.2　Entity Framework的基础知识

EF API是以前面两章所介绍的ADO.NET为基础的。和其他ADO.NET交互方式一样，Entity Framework使用一个ADO.NET数据提供程序来与数据存储进行交互。所不同的是，在能够与EF API交互之前，必须更新数据提供程序使其支持新的服务。微软SQL Server数据提供程序已经对底层结构进行了必要的更新以支持System.Data.Entity.dll程序集。

说明　很多第三方数据库（如Oracle和MySQL）提供了支持EF的数据提供程序。详情可以咨询你的数据库提供商或登录www.sqlsummit.com/dataprov.htm来查看已知的ADO.NET数据提供程序。

除了在微软SQL Server数据提供程序增加了必要的内容以外，System.Data.Entity.dll程序集还包含了各种命名空间来解释EF服务本身。EF API的两个关键部分为对象服务和实体客户端。

1. 对象服务的作用

对象服务是EF的一部分，它在代码中对客户端实体进行控制。例如，对象服务跟踪你对实体的更改（如将汽车的颜色由绿色改为蓝色）、管理实体间的关系（如查找用户Steve Hagen的所有订单）并提供将更改保存到数据库的方法，以及用XML或二进制序列化服务对实体状态进行持久化的方法。

就编程方面而言，对象服务层对所有扩展`EntityObject`基类的类进行管理。正如你所想的那样，`EntityObject`是EF编程模型中所有实体类的基类。例如，如果查找上例中`Car`实体的继承链，你会发现`Car`是一个`EntityObject`（如图23-4所示）。

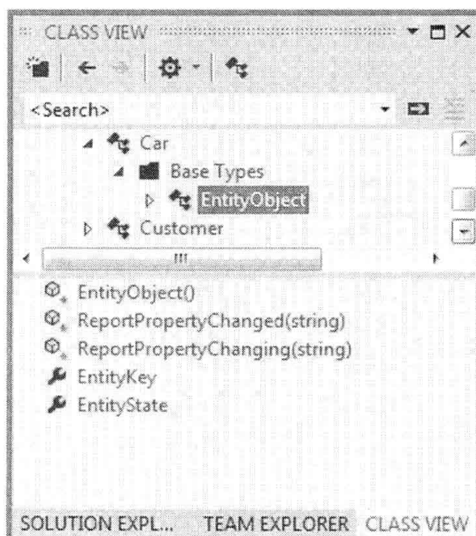


图23-4　EF的对象服务层将管理所有继承自`EntityObject`的类

2. 实体客户端的作用

EF API的另一个主要部分是实体客户端层，它使用基本的ADO.NET数据提供程序来建立数据库连接、基于实体状态和LINQ查询生成SQL语句、将数据库数据映射到实体，以及处理其他在不使用Entity Framework时常见的细节问题。

你可以在`System.Data.EntityClient`命名空间中浏览实体客户端层的功能。该命名空间包含一些类，可以将EF概念（如LINQ to Entity查询）映射到ADO.NET数据提供程序。这些类（如`EntityCommand`和`EntityConnection`）与ADO.NET数据提供程序中的类惊人地相似。例如，图23-5演示了实体客户端层是如何扩展与其他提供程序中相同的抽象基类的（如`DbCommand`和`DbConnection`，更详细的内容请参阅第21章）。

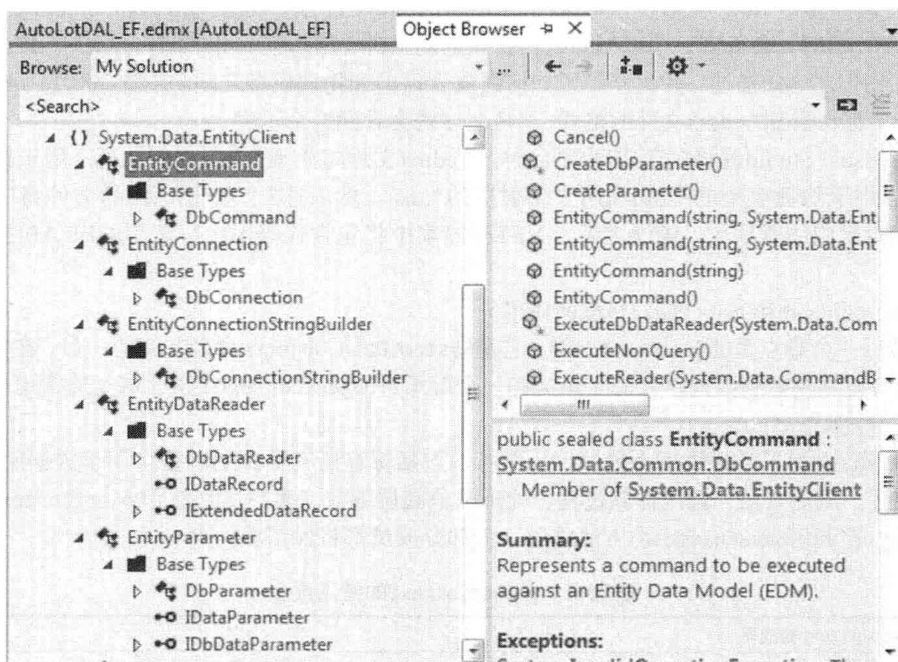


图23-5　实体客户端层将实体命令映射到基本的ADO.NET数据提供程序

实体客户端层一般在后台运行，但如果你想完全控制它的工作（如生成SQL查询和处理数据库返回的数据），也可以直接对其进行操作。

如果你需要更大程度地控制实体客户端基于LINQ查询创建SQL语句的方式，你可以使用Entity SQL。Entity SQL是直接作用于实体的与数据库无关的SQL方言。一旦创建了一个Entity SQL查询，它将被直接发送给实体客户端服务（也可以是对象服务），并被转换为符合基本数据提供程序的SQL语句。本章不会太深入地介绍Entity SQL，只是稍后会演示一些使用了这种新的基于SQL的语法示例。

如果你需要更大程度地控制获取记录集的操作方式，你可以不使用数据库结果在实体对象上的自动映射，而是使用`EntityDataReader`类手动处理这些数据。毫无疑问，`EntityDataReader`和`SqlDataReader`一样只允许对数据流进行只进的、只读的处理。稍后你将看到关于这种方法的示例。

3. *.edmx文件的作用

截止到目前，我们所介绍的实体是指客户端的类，其具有实体数据模型的功能。尽管客户端实体最终被映射为正确的数据库表，但实体类的属性名和数据表的列名之间却并非是紧耦合的。

为了使Entity Framework API能够将实体类数据正确映射到数据库表数据，你需要定义适当的映射逻辑。在所有数据模型驱动的系统中，实体、真正的数据库以及映射层都会被划分为3个相关的部分：概念模型、逻辑模型和物理模型。

- 概念模型定义了实体以及它们之间的关系（如果有的话）。
- 逻辑模型将实体和关系（通过外键约束）映射到表。
- 物理模型通过指定的存储细节（如表架构、表分割和索引）来表示特定的数据引擎的能力。

在EF的世界里，这三层均存放在基于XML格式的文件里。当使用Visual Studio集成的Entity Framework设计器时，会得到一个以*.edmx为扩展名的文件（EDM为entity data model）。该文件包含实体、物理数据库的XML描述，并且介绍了如何在概念模型和物理模型之间映射这些信息。在本章的第一个示例中，你将看到*.edmx文件的格式。（马上你就会看到这个示例。）

当使用Visual Studio编译基于EF的项目时，*.edmx文件将生成3个独立的文件：用于概念模型数据的*.csdl、用于物理模型的*.ssdl和用于映射层的*.msl。然后这3个基于XML的文件将以二进制资源的形式绑定到应用程序中。编译之后，.NET程序集中将包含代码库中所调用的EF API的所有必要数据。

4. ObjectContext和ObjectSet<T>类的作用

EF的最后一个难点是ObjectContext类，它是System.Data.Objects命名空间的一员。在生成*.emdx文件时，你将得到映射到数据库表的实体类和一个继承自ObjectContext的类。该类通常用于对象服务与实体客户端之间的交互。

ObjectContext为子类提供了大量的核心服务，包括保存所有更新的功能（用于数据库更新）、调整连接字符串、删除对象、调用存储过程、处理其他底层细节。表23-1描述了ObjectContext类的部分成员（注意，在调用SaveChanges()方法之前，大部分成员都将保留在内存中）。

表23-1　ObjectContext的常用成员

ObjectContext的成员	含　义
AcceptAllChanges()	接受对对象上下文中的实体对象所做的所有改变
AddObject()	向对象上下文中添加一个对象
DeleteObject()	对一个要删除的对象进行标记
ExecuteFunction<T>()	执行数据库中的一个存储过程
ExecuteStoreCommand()	直接向数据库发送一条SQL命令
GetObjectByKey()	通过主键在对象上下文中查询一个对象
SaveChanges()	向数据库提交所有更新
CommandTimeout	该属性为所有对象上下文操作获取或设置以秒记的超时值
Connection	该属性返回当前对象上下文使用的连接字符串
SavingChanges	当对象上下文向数据存储保存更改时将触发该事件

ObjectContext的派生类作为一个容器，管理那些存储在ObjectSet<T>集合中的实体对象。例如，如果为AutoLot数据库的Inventory表生成*.edmx文件，你（默认情况下）最终将得到一个AutoLotEntities类。该类包含一个Inventories属性（注意这里是复数形式），它封装了一个ObjectSet<Inventory>类型的数据成员。如果为AutoLot数据库的Orders表创建一个EDM，AutoLot-Entites类将定义另一个Orders属性，它封装了一个ObjectSet<Order>类型的成员变量。表23-2显示了System.Data.Objects.ObjextSet<T>的一些常用成员。

表23-2 ObjectSet<T>的常用成员

ObjectSet<T>的成员	含 义
AddObject()	向集合中插入一个新的实体对象
CreateObject<T>()	创建指定实体类型的实例
DeleteObject	标记一个要删除的对象

23

当获取了对象上下文中正确的属性后，就可以调用ObjectSet<T>的这些成员了。再次考虑本章前面提到的示例代码：

```
using (AutoLotEntities context = new AutoLotEntities())
{
  // 使用实体向Inventory表中添加一条新的记录
  context.Cars.AddObject(new Car() { AutoIDNumber = 987, CarColor = "Black",
                                     MakeOfCar = "Pinto",
                                     NicknameOfCar = "Pete" });
  context.SaveChanges();
}
```

这里的AutoLotEntities继承自ObjectContext。Cars属性用来访问ObjectSet<Car>变量。使用该引用插入一个新的Car实体对象，并通知ObjectContext将所有更改保存到数据库。

可以对ObjectSet<T>使用LINQ to Entity查询。这是因为ObjectSet<T>支持第12章中所介绍的那些扩展方法。此外，ObjectSet<T>还从它的直接父类ObjectQuery<T>中继承了大量的功能，后者代表一个强类型的LINQ（或Entity SQL）查询。

5. 汇总

图23-6显示了EF API的组织架构。在创建第一个Entity Framework示例之前，我们先来思考一下这幅图。

图23-6乍看上去似乎很复杂，其实不然。例如下面的场景：你对上下文中的实体编写了一个LINQ查询，该查询被传递给对象服务，对象服务将LINQ命令转换为实体客户端可以理解的树。然后，实体客户端将树转换为符合ADO.NET提供程序的SQL语句。提供程序返回一个数据读取器（如一个DbDataReader的派生对象），客户端服务使用该读取器（EntityDataReader）将数据传入对象服务。最终C#代码库所得到的是实体数据的枚举（IEnumerable<T>）。

还有一种情况，如果你的C#代码库希望对客户端服务创建的发送到数据库的SQL语句进行更多地控制，你可以编写C#代码直接将Entity SQL传递给实体客户端或对象服务。最终得到的也是一个IEnumerable<T>。

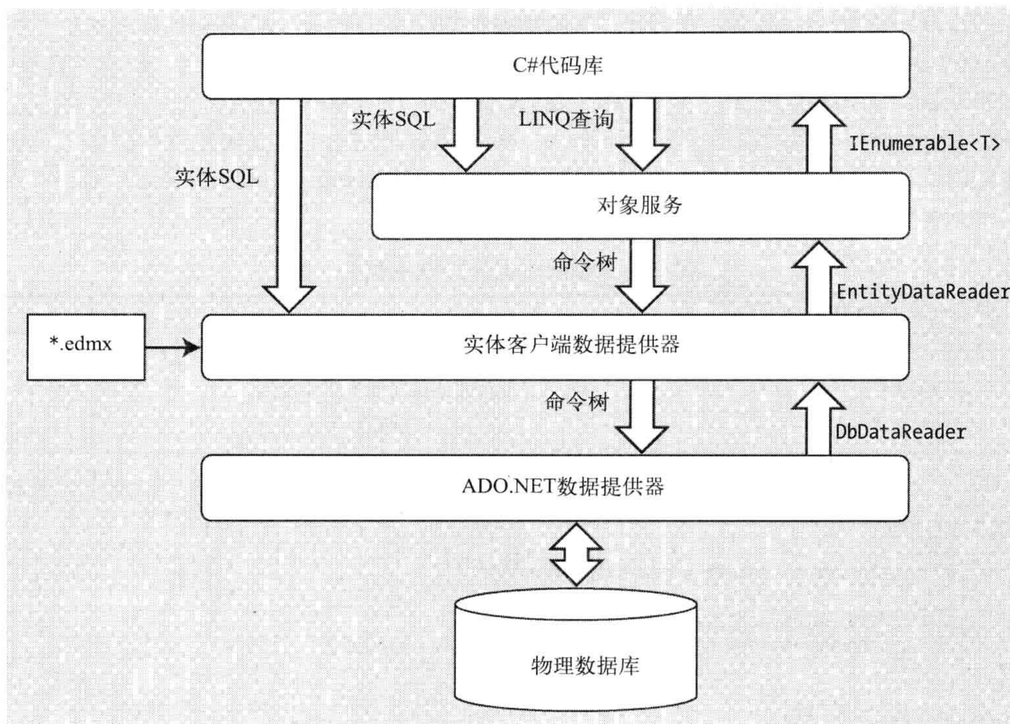


图23-6　ADO.NET Entity Framework的主要组件

在这两种情况下，你都必须使客户端服务理解*.edmx文件中的XML数据，即如何将数据库原子映射到实体。最后要记住的是，客户端（如C#代码库）还可以使用EntityDataReader直接从实体客户端获取数据。

23.2　创建和分析 EDM

我们已经对ADO.NET Entity Framework的功能和工作方式有了很深入的了解，现在来看一个完整的示例。简便起见，我们将创建一个只允许访问AutoLot库中Inventory表的EDM。在理解了这些基础知识之后，你可以为整个AutoLot库创建EDM，并在图形界面中显示这些数据。

1. 生成*.edmx文件

新建一个名为InventoryEDMConsoleApp的控制台应用程序。在使用Entity Framework时，第一个步骤就是创建由*.edmx文件定义的概念、逻辑、物理模型数据。可以使用.NET Framework 4.5 SDK命令行工具EdmGen.exe来创建*.edmx文件。打开Developer Command Prompt并输入如下指令：

```
EdmGen.exe -?
```

你会发现一个指令选项的列表，你可以使用这些选项为一个已有的数据库生成必要的文件；还有一些选项可以用来根据已知的实体文件生成新的数据库。表23-3列出了EdmGen.exe的一些常用选项。

表23-3 常用的EdmGen.exe命令行标记

EdmGen.exe选项	含 义
/mode:FullGeneration	为指定的数据库生成*.ssdl、*.msl、*.csdl文件和客户端实体
/project:	用来生成代码和文件的基本名称，通常为提取数据的数据库名称（可简写为/p:）
/connectionstring:	用来与数据库交互的连接字符串（可简写为/c:）
/language:	指定生成的代码为C#还是VB
/pluralize	只用英语语法规则，对实体集合名、实体类型名以及导航属性自动单数化或复数化

对于.NET 4.0来说，EF编程模型支持领域先行的编程，这允许我们先创建实体（典型的面向对象技术）然后生成全新的数据库。在本章对ADO.NET EF的介绍中，我们将不使用这种模型先行的思想，也不会使用EdmGen.exe生成客户端实体模型，而是使用Visual Studio中的图形化EDM设计器。

单击Project→Add New Item...菜单项，插入一个新的ADO.NET实体数据模型项InventoryEDM.edmx（记住要选择Data节点，如图23-7所示）。

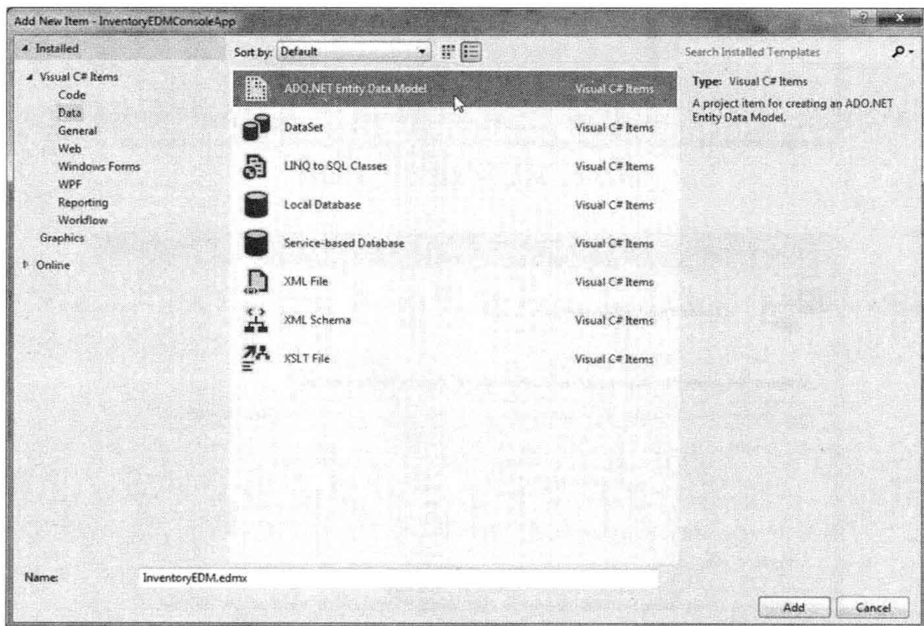


图23-7 为项目插入一个新的ADO.NET EDM项

单击Add按钮启动Entity Model Data向导。该向导第一个步骤是让你选择是从已有数据库生成EDM，还是定义一个空模型（为模型先行开发提供的支持）。选择Generate from database选项，单击Next按钮（如图23-8所示）。

在向导的第二个步骤中可以选择数据库。在下拉列表中将列出所有在Visual Studio Server Explorer中已有的数据库连接。如果下拉列表中没有，可以单击New Connection按钮。接下来选择AutoLot数据库，然后选择在（自动生成的）App.config文件中保存连接字符串数据（如图23-9所示）。

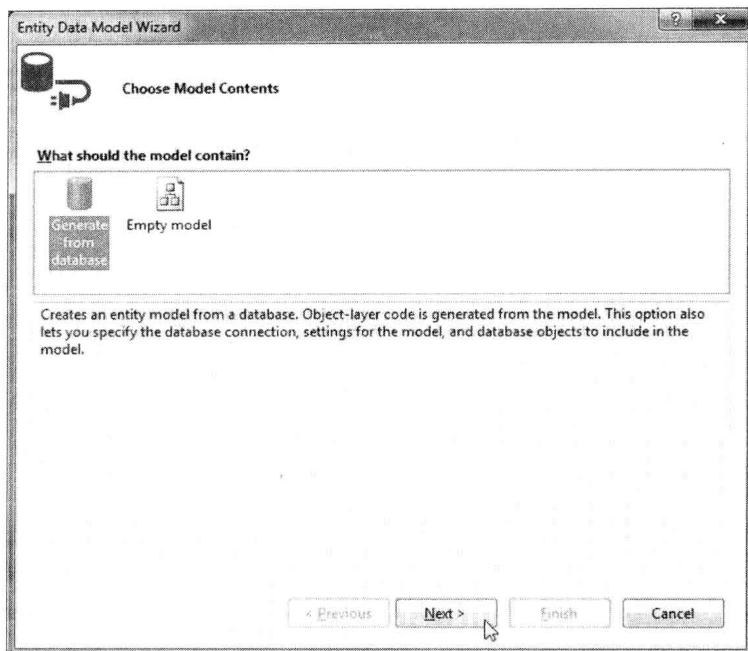


图23-8　从已知数据库生成EDM

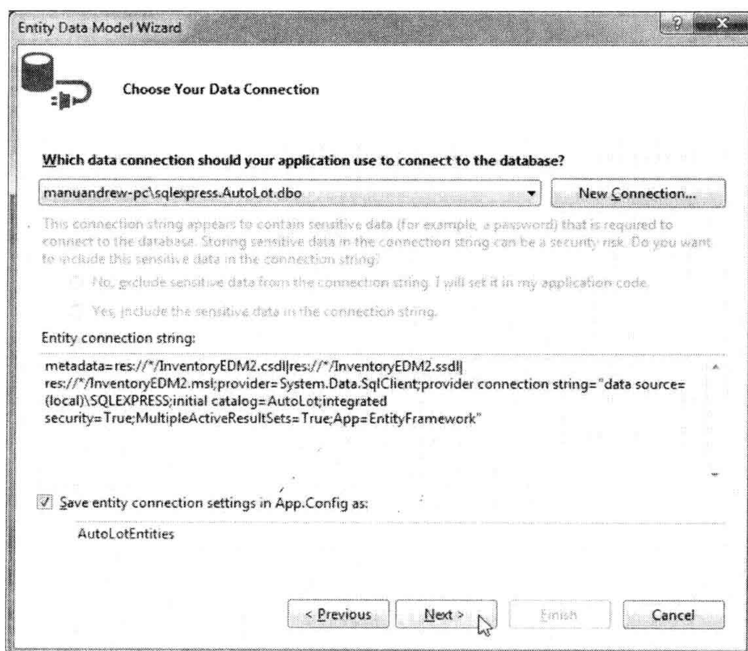


图23-9　选择要生成EDM的数据库

在单击Next按钮之前，先检查一下连接字符串的格式：

```
metadata=res://*/InventoryEDM.csdl|res://*/InventoryEDM.ssdl|res://*/InventoryEDM.msl;
  provider=System.Data.SqlClient;provider connection string=
  "Data Source=(local)\SQLEXPRESS;
  Initial Catalog=AutoLot;Integrated Security=True;Pooling=False"
```

由于机器配置不同，连接也会有所区别。注意这里的metadata标记，它用来标识概念文件、物理文件、映射文件等嵌入的XML资源数据的名称。（前面曾经介绍过，*.edmx文件在编译时将被拆分成3个独立的文件，这些文件中的数据将以二进制资源的形式嵌入到程序集中。）

在向导的最后一个步骤中，可以从数据库中选择要生成EDM的项。在本例中我们只关注Inventory表（如图23-10所示）。

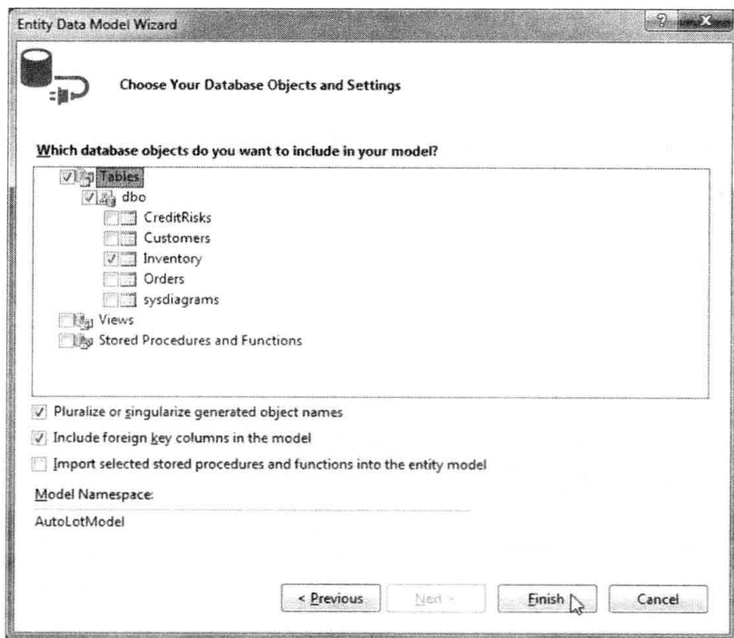


图23-10　选择数据库项

现在，单击Finish按钮生成EDM数据。

2. 改造实体数据

完成向导之后，IDE将打开EDM设计器，可以看到一个单独的实体Inventory。使用Entity Data Model Browser窗口可以浏览设计器中任意实体的结构（打开View→Other Windows菜单选项）。代表Inventory数据库表的概念模型位于Entity Types文件夹下（如图23-11所示），而该数据库的物理模型位于Store节点下，其具体名称取决于数据库本身的名称（本例中为AutoLotModel.Store）。

默认情况下，实体的名称取决于原始的数据库对象的名称。但概念模型中实体的名称是没有约束的。你可以更改实体名称及其属性的名称，即在设计器中选中一项，然后在Visual Studio的Properties窗口中修改Name属性的值。接下来，我们将Inventory实体改名为Car，将PetName属性改名为CarNickname（如图23-12所示）。

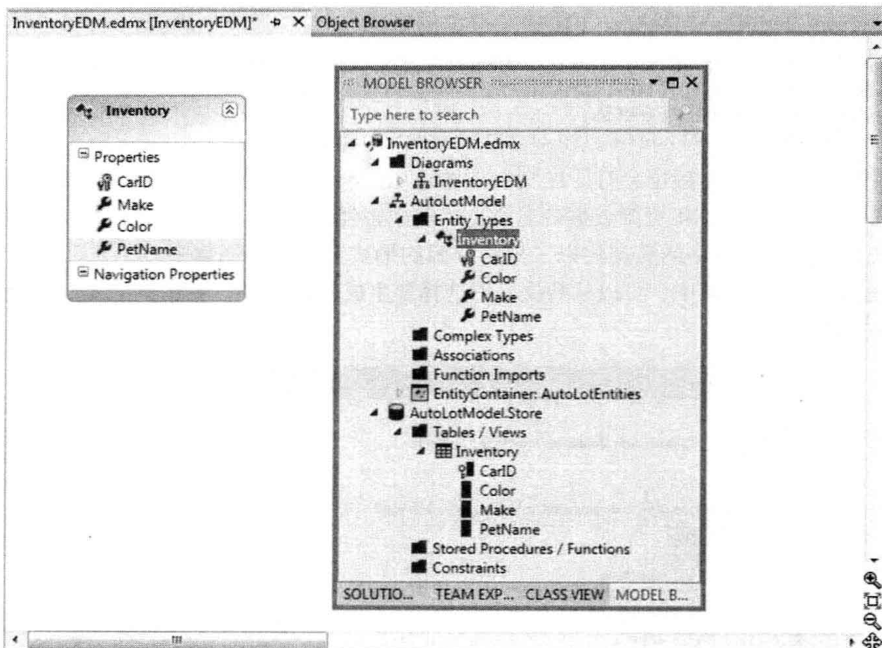


图23-11　Entity设计器和Model Browser窗口

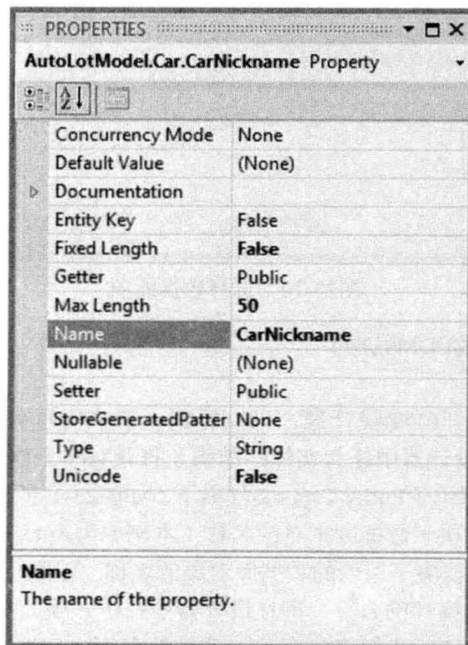


图23-12　在Properties窗口中修改实体

这时，概念模型如图23-13中所示。

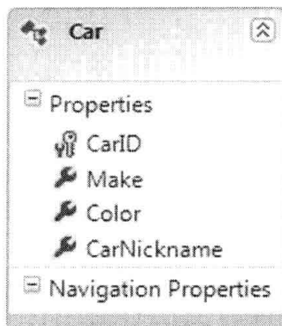


图23-13 客户端模型，为匹配业务领域而进行了改造

在设计器中选择整个Car实体，再查看Properties窗口。你会发现Entity Set Name字段也从Inventories改为Cars（如图23-14所示）。Entity Set的值是很重要的，因为它与数据上下文类的属性名相对应，而且可以用于修改数据库。ObjectContext派生类的这个属性封装了一个ObjectSet<T>成员变量。

编译应用程序，然后刷新代码库，可以看到基于*.edmx文件数据生成的*.csdl、*.msdl、*.ssdl文件。

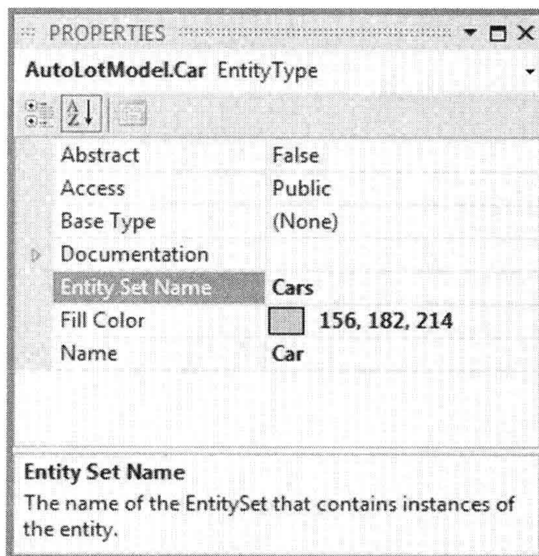


图23-14 封装ObjectSet<T>的属性名称

3. 查看映射

修改了数据之后，你可以通过Mapping Details窗口（打开View→Other Windows...菜单项）来浏览概念层和物理层之间的映射。如图23-15所示，树左侧的节点代表物理层中的数据名称，而右侧节点代表概念模型的名称。

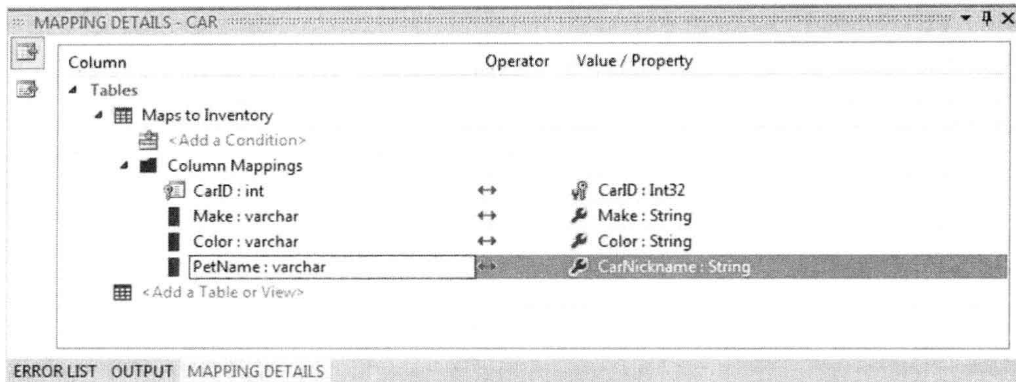


图23-15　Mapping Details窗口显示了概念模型和物理模型的映射

4. 查看生成的*.edmx文件的数据

现在我们来看看EDM向导为我们生成了什么。在Solution Explorer中右击InventoryEDM.edmx文件，选择Open With...菜单项。在弹出的对话框中选择XML (Text) Editor选项。这样我们就可以在EDM设计器中查看实际的XML数据。该XML文档的结构位于<edmx:Edmx>根元素内。

该根元素包含两个子元素。<edmx:Runtime>包含应用程序在运行时使用的元数据，<Designer>包含Visual Studio在开发时使用的元数据。

在<edmx:Runtime>元素内包含三个子元素，分别描述物理存储模型、逻辑C#对象模型以及它们之间的映射。我们先来看看存储模型元数据。

```
<!-- SSDL内容-->
<edmx:StorageModels>
  <Schema Namespace="AutoLotModel.Store" Alias="Self"
   Provider="System.Data.SqlClient"
   ProviderManifestToken="2008"
   xmlns:store=
   "http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
   xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
   <EntityContainer Name="AutoLotModelStoreContainer">
     <EntitySet Name="Inventory" EntityType="AutoLotModel.Store.Inventory"
                store:Type="Tables" Schema="dbo" />
   </EntityContainer>
   <EntityType Name="Inventory">
     <Key>
       <PropertyRef Name="CarID" />
     </Key>
      <Property Name="CarID" Type="int" Nullable="false" />
      <Property Name="Make" Type="varchar" Nullable="false" MaxLength="50" />
      <Property Name="Color" Type="varchar" Nullable="false" MaxLength="50" />
      <Property Name="PetName" Type="varchar" MaxLength="50" />
    </EntityType>
  </Schema>
</edmx:StorageModels>
```

注意，<Schema>节点中定义了ADO.NET数据提供程序的名称，在与数据库(System.Data.SqlClient)通信时会用到这些信息。<EntityType>节点标记了物理数据库表的名称以及表中各列的名称。

*.edmx文件中下一个重要的部分是<edmx:ConceptualModels>元素，它定义了更改过的客户端实体。注意，Cars实体定义了CarNickname属性，这是我们在设计器中修改过的：

```
<!--CSDL内容-->
<edmx:ConceptualModels>
  <Schema Namespace="AutoLotModel" Alias="Self"
    xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
    xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
    <EntityContainer
      Name="AutoLotEntities" annotation:LazyLoadingEnabled="true">
      <EntitySet Name="Cars" EntityType="AutoLotModel.Car" />
    </EntityContainer>
    <EntityType Name="Car">
      <Key>
        <PropertyRef Name="CarID" />
      </Key>
        <Property Name="CarID" Type="Int32" Nullable="false" />
        <Property Name="Make" Type="String" Nullable="false" MaxLength="50"
                  Unicode="false" FixedLength="false" />
        <Property Name="Color" Type="String" Nullable="false" MaxLength="50"
                  Unicode="false" FixedLength="false" />
        <Property Name="CarNickname" Type="String" MaxLength="50"
                  Unicode="false" FixedLength="false" />
    </EntityType>
  </Schema>
</edmx:ConceptualModels>
```

这之后是映射层，Mapping Details窗口（以及EF运行时）使用映射层来联系概念模型和物理模型的名称：

```
<!-- C-S映射内容-->
<edmx:Mappings>
  <Mapping Space="C-S"
    xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
  <EntityContainerMapping StorageEntityContainer="AutoLotModelStoreContainer"
    CdmEntityContainer="AutoLotEntities">
    <EntitySetMapping Name="Cars">
      <EntityTypeMapping TypeName="AutoLotModel.Car">
        <MappingFragment StoreEntitySet="Inventory">
          <ScalarProperty Name="CarID" ColumnName="CarID" />
          <ScalarProperty Name="Make" ColumnName="Make" />
          <ScalarProperty Name="Color" ColumnName="Color" />
          <ScalarProperty Name="CarNickname" ColumnName="PetName" />
        </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
    </EntityContainerMapping>
  </Mapping>
</edmx:Mappings>
```

*.edmx文件的最后一部分是<Designer>元素，EF运行时并不使用这些数据。查看该数据你会发现它包含了一些指令，Visual Studio使用这些指令在可视的设计器界面上显示实体。

确保项目至少被编译过一次，然后单击Solution Explorer中的Show All Files按钮。然后打开obj\Debug文件夹，进入edmxResourcesToEmbed子目录。在该目录下你会发现基于整个*.edmx文件而生成的3个XML文件（如图23-16所示）。

这些文件中的数据将以二进制的形式嵌入到程序集中。因此，.NET应用程序具备所有用于理解EDM中概念、物理和映射层的信息。

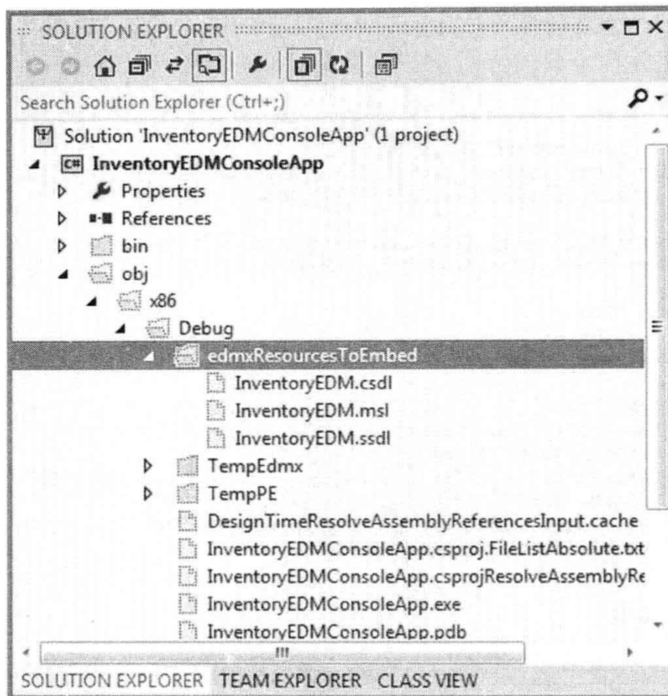


图23-16　使用*.edmx文件生成3个独立的XML文件

5. 查看生成的源代码

你已经准备好用EDM来编写一些代码了。但在这之前你应该检查一下生成的C#代码。打开Class View窗口，展开默认的命名空间。你会发现除了Program类以外，EDM向导还生成了一个实体类（被更名为Car）以及AutoLotEntities类。

如果回到Solution Explorer展开InventoryEDM.edmx节点，你会发现一个用于IDE维护的文件InventoryEDM.Designer.cs。和其他用于IDE维护的文件一样，你不能直接编辑该文件，因为在每次编译时IDE都会重新创建该文件。但你可以双击来浏览它的内容。

AutoLotEntites类扩展了EF编程模型的入口点ObjectContext类。AutoLotEntities的构造函数提供了 多种方式来输入连接字符串数据。默认的构造函数将自动从App.config文件读取连接字符串数据：

```
public partial class AutoLotEntities : ObjectContext
{
  public AutoLotEntities() : base("name=AutoLotEntities", "AutoLotEntities")
  {
    this.ContextOptions.LazyLoadingEnabled = true;
    OnContextCreated();
  }
...
}
```

注意，AutoLotEntites类的Cars属性封装了ObjectSet<Car>数据成员。你可以使用该属性访问EDM模型，来间接修改后台数据库：

```
public partial class AutoLotEntities : ObjectContext
{
...
  public ObjectSet<Car> Cars
  {
    get
    {
      if ((_Cars == null))
      {
        _Cars = base.CreateObjectSet<Car>("Cars");
      }
      return _Cars;
    }
  }
  private ObjectSet<Car> _Cars;
}
```

说明 你还会在ObjectContext派生类中看到一些以AddTo开头的方法，可以用来向ObjectSet<T>成员变量中添加新的实体。不过推荐的方法是使用通过强类型属性得到的ObjectSet<T>成员。

最后在设计器代码文件中还有一个有趣的东西就是Car实体类。实体类中的大部分代码都是构成概念模型的属性集合。每个属性的set逻辑都调用了EF API的StructuralObject.SetValidValue()静态方法。

并且，set逻辑中的代码还会将实体状态的改变通知给EF运行时。这是十分重要的，因为ObjectContext必须知道这些改变，并将它们更新到物理数据库。

此外，set的逻辑中还调用了两个分部方法。C#分部方法提供了在应用程序中处理更改通知的简单方式。如果没有实现分部方法，编译器将忽略这个调用。下面是Car实体类的CarNickname属性的实现：

```
public partial class Car : EntityObject
{
...
  public global::System.String CarNickname
  {
    get
    {
      return _CarNickname;
    }
    set
    {
      OnCarNicknameChanging(value);
      ReportPropertyChanging("CarNickname");
      _CarNickname = StructuralObject.SetValidValue(value, true);
      ReportPropertyChanged("CarNickname");
      OnCarNicknameChanged();
    }
  }
  private global::System.String _CarNickname;
  partial void OnCarNicknameChanging(global::System.String value);
  partial void OnCarNicknameChanged();
}
```

6. 强化生成的代码

设计器生成的所有的类都使用partial关键字声明，这允许我们在多个C#代码文件中实现这些类。这在使用EF编程模型时是非常有用的，因为这意味着你可以为实体类添加“真正”的方法，从而更好

地对业务领域进行建模。

在本例中，我们重写Car实体类的ToString()方法，返回一个格式化的字符串来描述实体的状态。我们还完成了部分方法OnCarNicknameChanging()和OnCarNicknameChanged()的定义，可以完成简单的通知操作。我们在新的Car.cs文件中定义的部分类声明如下：

```
public partial class Car
{
  public override string ToString()
  {
    // 由于PetName列可能为空字符串，因此提供默认值“**No Name**”
    return string.Format("{0} is a {1} {2} with ID {3}.",
      this.CarNickname ?? "**No Name**",
      this.Color, this.Make, this.CarID);
  }
  partial void OnCarNicknameChanging(global::System.String value)
  {
    Console.WriteLine("\t-> Changing name to: {0}", value);
  }
  partial void OnCarNicknameChanged()
  {
    Console.WriteLine("\t-> Name of car has been changed!");
  }
}
```

友情提示：在实现这些方法之后，你得到的通知是指实体类的属性已经被修改或将要被修改，而不是指物理数据库已经被修改。如果你需要知道物理数据库是否被修改，可以使用ObjectContext派生类的SavingChanges事件处理程序。

23.3 对概念模型进行编程

现在我们可以写一些与EDM交互的代码了。修改Program类，在Main()方法中调用两个辅助方法。其中一个辅助方法使用概念模型打印Inventory数据库表的所有记录，另一个辅助方法向Inventory表中插入一条新的记录，如下所示：

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with ADO.NET EF *****\n");
    AddNewRecord();
    PrintAllInventory();
    Console.ReadLine();
  }

  private static void AddNewRecord()
  {
    // 向AutoLot数据库的Inventory表添加一条记录
    using (AutoLotEntities context = new AutoLotEntities())
    {
      try
      {
        // 对新的记录进行硬编码，仅供测试
        context.Cars.AddObject(new Car() { CarID = 2222,
             Make = "Yugo", Color = "Brown" });
        context.SaveChanges();
```

```
            }
            catch(Exception ex)
            {
              Console.WriteLine(ex.InnerException.Message);
            }
          }
        }

        private static void PrintAllInventory()
        {
          // 选择AutoLot中Inventory表的所有记录，并使用Car实体类的自定义ToString()方法打印其数据
          using (AutoLotEntities context = new AutoLotEntities())
          {
            foreach (Car c in context.Cars)
              Console.WriteLine(c);
          }
        }
      }
```

23

在前面你看到过类似的代码，不过现在你应该已经更清楚其工作原理了。每个辅助方法都创建了一个ObjectContext派生类（AutoLotEntities）实例。并使用强类型的Cars属性来与ObjectSet<Car>字段进行交互。在对Cars属性进行枚举时，将会间接向ADO.NET数据提供程序提交一条SQL SELECT语句。在使用ObjectSet<Car>的AddObject()方法插入新的Car对象然后调用上下文的SaveChanges()方法时，你实际上执行了一条SQL INSERT语句。

1. 删除记录

当你想在数据库中删除某条记录时，首先要在ObjectSet<T>中找到正确的项，这可以使用GetObjectByKey()方法，并传递一个EntityKey对象（位于System.Data命名空间）。假设已经在C#代码文件中引入了该命名空间，辅助方法的代码如下：

```
private static void RemoveRecord()
{
  // 通过主键查找要删除的汽车
  using (AutoLotEntities context = new AutoLotEntities())
  {
    // 为查找的实体定义主键
    EntityKey key = new EntityKey("AutoLotEntities.Cars", "CarID", 2222);

    // 查找实体，如果存在的话将其删除
    Car carToDelete = (Car)context.GetObjectByKey(key);
    if (carToDelete != null)
    {
      context.DeleteObject(carToDelete);
      context.SaveChanges();
    }
  }
}
```

说明 无论是好是坏，调用GetObjectByKey()会在删除对象之前访问一次数据库。

注意，在创建EntityKey对象时，你需要使用一个string对象通知构造函数使用ObjectContext派生类的哪个ObjectSet<T>。第二个参数也是string，表示实体类中标记为主键的属性名称。构造函数的最后一个参数是主键的值。找到要删除的对象后，调用上下文的DeleteObject()方法，然后保存更改。

2. 更新记录

更新记录也是很简单的。先定位对象，设置其属性值，然后保存更改，如下所示：

```
private static void UpdateRecord()
{
  // 通过主键查找要更新的汽车
  using (AutoLotEntities context = new AutoLotEntities())
  {
    // 为查找的实体定义主键
    EntityKey key = new EntityKey("AutoLotEntities.Cars", "CarID", 2222);

    // 获取实体，更改并保存
    Car carToUpdate = (Car)context.GetObjectByKey(key);
    if (carToUpdate != null)
    {
      carToUpdate.Color = "Blue";
      context.SaveChanges();
    }
  }
}
```

这个方法看上去有点奇怪，但如果你意识到GetObjectByKey()方法返回的实体对象是ObjectSet<T>字段中一个某个对象的引用的话，就不会再有这种疑虑了。在你设置属性改变实体状态的时候，你也在更改内存中的同一个对象。

说明　和ADO.NET的DataRow对象一样，任何EntityObject的子类（即所有实体类）都包含一个EntityState属性。对象上下文使用该属性来判断该实体是否被修改、删除、拆分（detach），等等。这是使用这种编程模型的默认设置，当然你可以根据需要手动进行修改。

---

3. 用LINQ to Entites进行查询

到目前为止，我们已经通过一些关于对象上下文和实体对象的示例学习了如何进行查询、插入、更新和删除。这都是很有用的，但其实EF在和LINQ查询一起使用的时候才会更强大。如果打算使用LINQ来更新或删除记录，你就无需再手动创建一个EntityKey对象。考虑下面这个修改后的RemoveRecord()方法，它将不会按照预期那样工作：

```
private static void RemoveRecord()
{
  // 通过主键查找要删除的汽车
  using (AutoLotEntities context = new AutoLotEntities())
  {
    // 判断实体是否存在
    var carToDelete = from c in context.Cars where c.CarID == 2222 select c;

    if (carToDelete != null)
    {
      context.DeleteObject(carToDelete);
      context.SaveChanges();
    }
  }
}
```

这段代码可以通过编译，但在调用DeleteObject()方法时会抛出运行时异常。理由是LINQ查询返回的是ObjectQuery<T>对象，而不是Car对象。要记住使用LINQ查询定位实体时，得到的是一个

ObjectQuery<T>对象，它代表能够返回所需数据的查询。要执行查询（并返回Car实体），就必须调用查询对象的某个方法，如FirstOrDefault()。示例如下：

```
var carToDelete =
  (from c in context.Cars where c.CarID == 2222 select c).FirstOrDefault();
```

通过调用ObjectQuery<T>的FirstOrDefault()可以找到所需的项，而如果没有ID为2222的Car，那么默认的值为null。

由于在第13章已经学习了大量的LINQ表达式，再举几个例子应该就足够了：

```
private static void FunWithLINQQueries()
{
  using (AutoLotEntities context = new AutoLotEntities())
  {
    // 获取新数据的投影
    var colorsMakes = from item in context.Cars select
        new { item.Color, item.Make };
    foreach (var item in colorsMakes)
    {
      Console.WriteLine(item);
    }

    // 只获取CarID < 1000的记录
    var idsLessThan1000 = from item in context.Cars
        where item.CarID < 1000 select item;
    foreach (var item in idsLessThan1000)
    {
      Console.WriteLine(item);
    }
  }
}
```

虽然这些查询的语法很简单，但要记住每当你对对象上下文执行一次LINQ查询时都会访问一次数据库。当你希望对数据执行LINQ查询并得到一个单独的副本时，你需要使用ToList<T>()、ToArray<T>()、ToDictionary<K,V>()等扩展方法来立即执行这些查询。下面的代码修改了前面的方法，它执行一个等价的SELECT *语句，将实体缓存到数组中，然后使用LINQ to Object对数组进行操作：

```
using (AutoLotEntities context = new AutoLotEntities())
{
  // 获取Inventory表中的所有数据
  // 也可以写成
  //var allData = (from item in context.Cars select ite).ToArray();
  var allData = context.Cars.ToArray();

  // 获取新数据的投影
  var colorsMakes = from item in allData select new { item.Color, item.Make };

  // 只获取CarID < 1000的记录
  var idsLessThan1000 = from item in allData where
    item.CarID < 1000 select item;
}
```

如果EDM中包含多个相关的表，使用LINQ to Entity将更有吸引力，稍后将看到这方面的示例。不过这个示例将暂时告一段落。我们来看看与对象上下文交互的其他两种方式。

4. 使用Entity SQL进行查询

可以肯定的是，大部分时间你都会使用LINQ来查询ObjectSet<T>。实体客户端会将你的LINQ查

询转换成适合的SQL语句，并将其传递给数据库进行处理。但如果你想对查询的格式有更多的控制，可以使用Entity SQL。

Entity SQL是一门可用于实体的类SQL查询语言。虽然Entity SQL语句的格式与传统的SQL语句十分类似，但他们其实是不同的。Entity SQL有独特的语法，这是因为查询的对象是实体而不是物理数据库。与LINQ to Entity查询类似，Entity SQL查询也将生成“真正”的SQL查询。

本章只介绍一个简单的示例，而不会深入介绍创建Entity SQL命令的细节，更详细的内容请参阅.NET Framework 4.5 SDK文档。考虑下面的方法，它创建了一个Entity SQL语句，用来查找ObjectSet<Car>集合中所有黑色的轿车：

```
private static void FunWithEntitySQL()
{
  using (AutoLotEntities context = new AutoLotEntities())
  {
    // 构建一个包含Entity SQL语法的字符串
    string query = "SELECT VALUE car FROM AutoLotEntities.Cars " +
                   "AS car WHERE car.Color='black'";

    // 现在基于该字符串构建一个ObjectQuery<T>
    var blackCars = context.CreateQuery<Car>(query);
    foreach (var item in blackCars)
    {
      Console.WriteLine(item);
    }
  }
}
```

注意，我们将一个格式化的Entity SQL语句作为参数传递给了对象上下文的CreateQuery<T>方法。

5. 使用实体客户端的数据阅读器对象

当使用LINQ to Entity或Entity SQL时，得到的数据将被自动映射为实体类，这要感谢实体客户端服务。通常情况下这就是我们想要的，然而你还可以使用EntityDataReader在结果集合映射到实体对象之前拦截它们，并进行手工处理。

下面看到的是本示例最后一个辅助方法，它使用了System.Data.EntityClient命名空间下的一些成员，通过一个命令对象和数据阅读器，手工建立了一个数据连接。这段代码与第21章中的类似，最大的不同是这里使用了Entity SQL，而不是“普通的”SQL。

```
private static void FunWithEntityDataReader()
{
  // 基于*.config文件创建一个连接对象
  using (EntityConnection cn = new EntityConnection("name=AutoLotEntities"))
  {
    cn.Open();

    // 构建一个Entity SQL查询
    string query = "SELECT VALUE car FROM AutoLotEntities.Cars AS car";

    // 创建一个命令对象
    using (EntityCommand cmd = cn.CreateCommand())
    {
      cmd.CommandText = query;

      // 最后，获取数据阅读器并处理得到的记录
      using (EntityDataReader dr =
```

```
                cmd.ExecuteReader(CommandBehavior.SequentialAccess))
          {
            while (dr.Read())
            {
              Console.WriteLine("***** RECORD *****");
              Console.WriteLine("ID: {0}", dr["CarID"]);
              Console.WriteLine("Make: {0}", dr["Make"]);
              Console.WriteLine("Color: {0}", dr["Color"]);
              Console.WriteLine("Pet Name: {0}", dr["CarNickname"]);
              Console.WriteLine();
            }
          }
        }
      }
    }
```

很好！这个初步的示例将对你理解Entity Framework的细节大有裨益。如前所述，如果EDM包含关联的数据表将会更加有趣。接下来我们就来学习这方面的内容。

23

源代码 InventoryEDMConsoleApp示例的源代码位于Chapter 23子目录下。

23.4 AutoLotDAL 4.0 版，加入实体

接下来我们将学习如何创建包含AutoLot数据库大部分内容和GetPetName存储过程的EDM。我强烈建议你复制一份第22章创建的AutoLotDAL(Version Three)，并将其命名为AutoLotDAL(Version Four)。

在Vusial Studio中打开最新版的AutoLotDAL项目，插入一个新的ADO.NET Entity Data Model项并将其命名为AutoLotDAL_EF.edmx。在向导的第三步中，我们选择`Inventory`、`Orders`、`Customers`表（这里暂时没必要选择CreditRisks表），以及自定义的存储过程（如图23-17所示）。

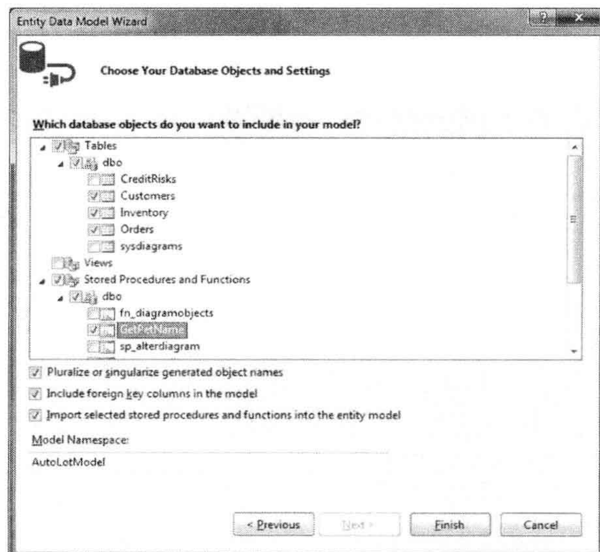


图23-17 包含AutoLot数据库大部分内容的*.edmx文件

与第一个EDM示例所不同的是，这次我们不必修改实体类的名称及其属性。无论何时打开Model Browser窗体，除了自定义存储过程外，还可以看到所有实体（如图23-18所示）。

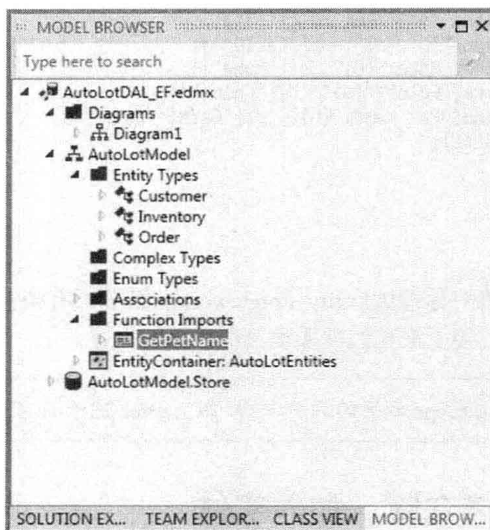


图23-18　导入的模型数据

23.4.1　导航属性的作用

观察EDM设计器会发现已经包含所有选中的表了，并且实体类的Navigation Properties节点下还多了新的条目（如图23-19所示）。

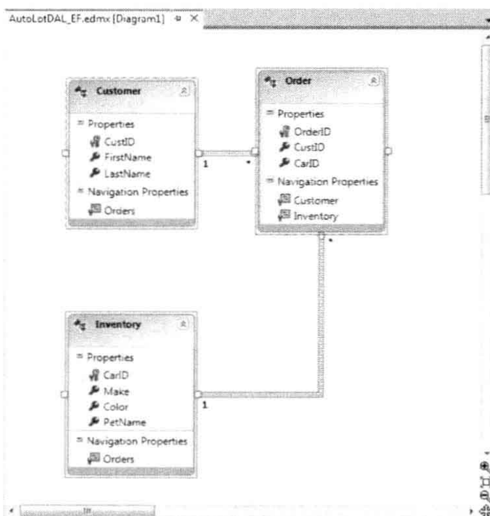


图23-19　导航属性

顾名思义，导航属性使我们可以在Entity Framework编程模型中实现JOIN操作（无需复杂的SQL语句）。为了说明这些外键关系，*.edmx文件中的每个实体现在都增加了新的XML数据，来说明这些实体是如何通过键数据进行关联的。如果你想直接看到这些标记，可以在XML编辑器中打开*.edmx文件。你也可以在Model Browser窗口的Associations文件夹下查看相同的信息（如图23-20所示）。

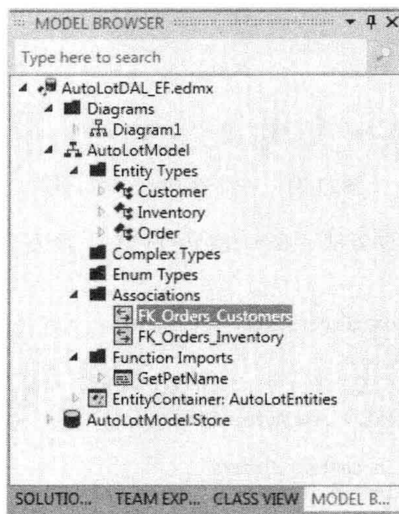


图23-20 查看实体关系

如果你愿意，可以将这个新库的版本号改为4.0.0.0（使用Properties窗口Applications选项卡中的Assembly Information按钮）。编译修改后的AutoLotDAL.dll程序集，然后将其移动到第一个客户端应用程序中。

源代码 AutoLotDAL（Version 4）的源代码位于Chapter 23子目录中。

23.4.2 在LINQ to Entity查询中使用导航属性

接下来我们将学习如何在LINQ to Entity查询中使用导航属性（导航属性也可用于Entity SQL，不过本书不会深入展开这个话题）。在将数据绑定到Windows Forms GUI之前，我们先创建一个控制台应用程序AutoLotEDMClient，并添加System.Data.Entity.dll和最新版的AutoLotDAL.dll的引用。

下一步，打开AutoLotDAL（Version Four）项目中的App.config文件（使用Project→Open…→File菜单项），将连接字符串复制到当前配置文件（如果项目中还没有App.config，可以通过Project→Add Existing Item添加整个文件）。同时，在初始的C#代码文件中引入`AutoLotDAL`命名空间。

然后，向`Orders`物理表添加一些新的记录，并且要保证某个客户有多个订单。打开Visual Studio的Server Explorer，向`Orders`表添加几条新记录，确保某个客户有2个以上的订单。例如，在图23-21中，编号为#4的客户有2个订单，分别是汽车#1992和#83。

Inventory: Query(m...sqlexpress.AutoLot)　Orders: Query(manu...qlexpress.AutoLot)

	OrderID	CustID	CarID
	1000	1	1000
	1001	2	32
	1002	3	888
	1003	4	1992
▶	1005	4	83
*	NULL	NULL	NULL

5 of 5

图23-21　一个客户有多个订单

现在在Program中添加一个辅助方法（在Main()中调用）。该方法使用导航属性来获取某个客户订单中的所有Inventory对象：

```
private static void PrintCustomerOrders(string custID)
{
  int id = int.Parse(custID);

  using (AutoLotEntities context = new AutoLotEntities())
  {
    var carsOnOrder = from o in context.Orders
      where o.CustID == id select o.Inventory;
    Console.WriteLine("\nCustomer has {0} orders pending:", carsOnOrder.Count());
    foreach (var item in carsOnOrder)
    {
      Console.WriteLine("-> {0} {1} named {2}.",
        item.Color, item.Make, item.PetName);
    }
  }
}
```

运行应用程序，输出结果如下（注意，在Main()中调用PrintCustomerOrders()时传入的客户ID为4）：

```
***** Navigation Properties *****
Please enter customer ID: 4

Customer has 2 orders pending:
-> Pink Saab named Pinky.
-> Rust Ford named Rusty.
```

在本例中，我们先查找上下文中CustID为指定值的Customer实体。找到之后，导航到Inventory表查找订单中的汽车。LINQ查询的返回值是一个Inventory对象的枚举，然后使用标准的foreach循环进行打印。

23.4.3　调用存储过程

如果在AutoLotDAL EMD中需要调用GetPetName存储过程，可以使用如下所示的两种方法之一。完整的代码清单如下所示：

```
private static void CallStoredProc()
{
  using (AutoLotEntities context = new AutoLotEntities())
  {
    // 方法 #1
    ObjectParameter input = new ObjectParameter("carID", 83);
    ObjectParameter output = new ObjectParameter("petName", typeof(string));

    // 调用上下文的ExecuteFunction方法
    context.ExecuteFunction("GetPetName", input, output);

    // 方法 #2
    // 或使用上下文中强类型的方法
    context.GetPetName(83, output);

    Console.WriteLine("Car #83 is named {0}", output.Value);
  }
}
```


在这段代码中可以看到，第一个方法是调用对象上下文的ExecuteFunction()方法，存储过程的名称由一个字符串指定，参数由ObjectParameter类型的对象表示。ObjectParameter类型位于System.Data.Objects命名空间，要记得在C#代码文件中引用该命名空间。

而第二个方法（同样请看上面的代码）是在对象上下文中使用强类型的名称。该方法要简单一些，所输入的参数（如carID）为强类型数据，而不是ObjectParameter对象。

源代码 AutoLotEDMClient示例的源代码位于Chapter 23子目录下。

23.5 将数据实体绑定到 Windows Forms GUI

要介绍ADO.NET Entity Framework的这一部分，我们首先创建一个简单的示例。在该示例中，我们将实体对象绑定到Windows Forms GUI中。如本章前面所述，你还可以将数据绑定操作用于WPF和ASP.NET项目。

我们创建一个名为AutoLotEDM_GUI的Windows Forms应用程序，将原始窗体名称改为MainForm.cs。然后添加对System.Data.Entity.dll和最新版的AutoLotDAL.dll的引用。最后修改这个新项目中的App.config文件，添加到AutoLotDAL（Version Four）的连接字符串，并在窗体的主代码文件中引入AutoLotDAL命名空间。

现在在窗体设计器中添加一个DataGridView对象，将其名称改为gridInventory。重命名之后，选择内置的grid编辑器（右上方的小箭头），在Choose Data Source下拉框中添加项目的数据源（如图23-22所示）。

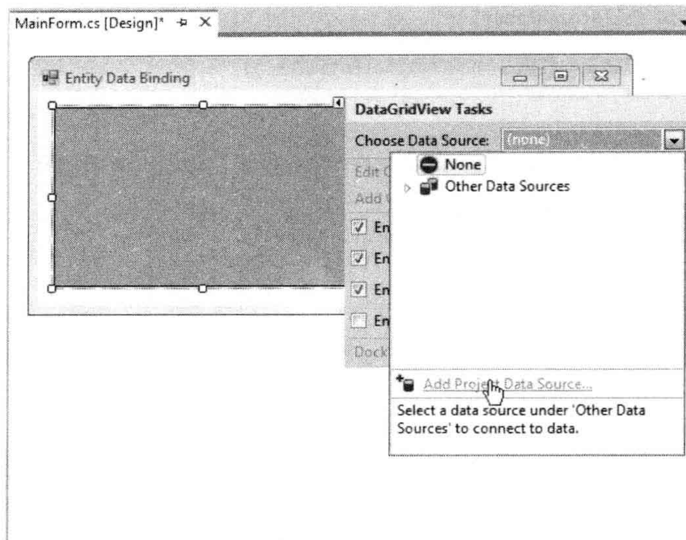


图23-22　设计Windows Forms DataGridView控件

在本例中，我们并不是直接绑定数据库，而是绑定实体类，因此选择Object选项（如图23-23所示）。

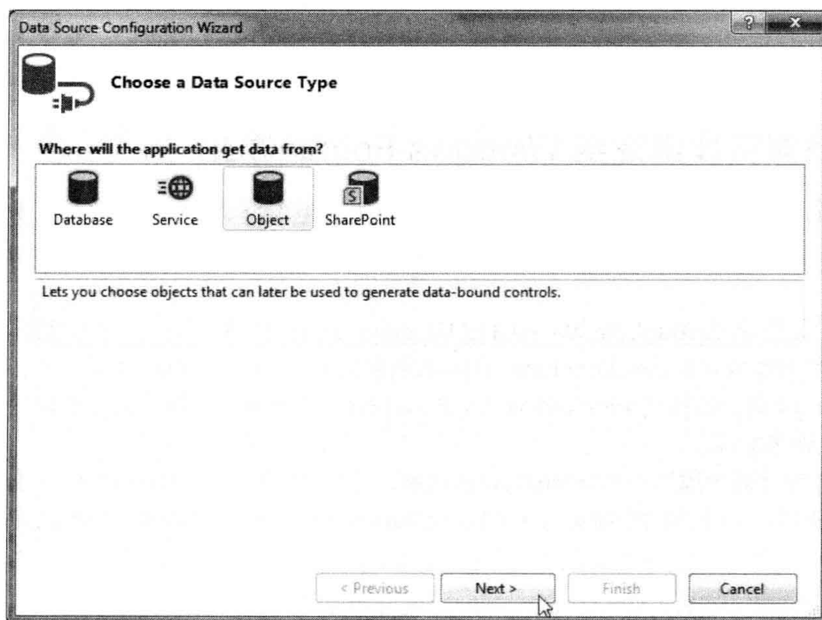


图23-23　绑定强类型对象

在最后一个步骤中，选中AutoLotDAL.dll下的Inventory表，如图23-24所示（如果没有看到该项，则很可能是忘记了添加对该库的引用）。

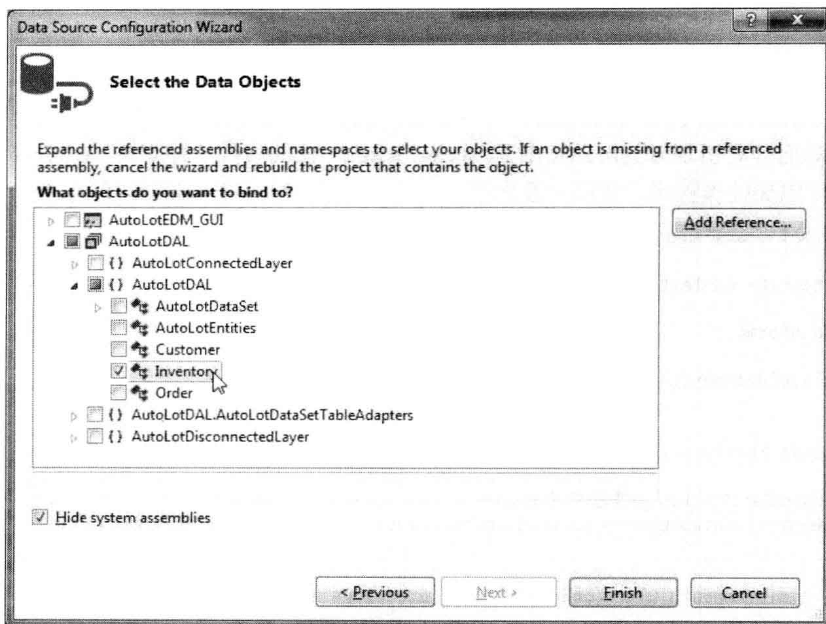


图23-24　选择Inventory表

单击Finish按钮，你将看到grid显示了Inventory实体类的各个属性，包括那些导航属性。完成之后，添加一个Button控件，并改名为btnUpdate。这时你的设计器将如图23-25所示。

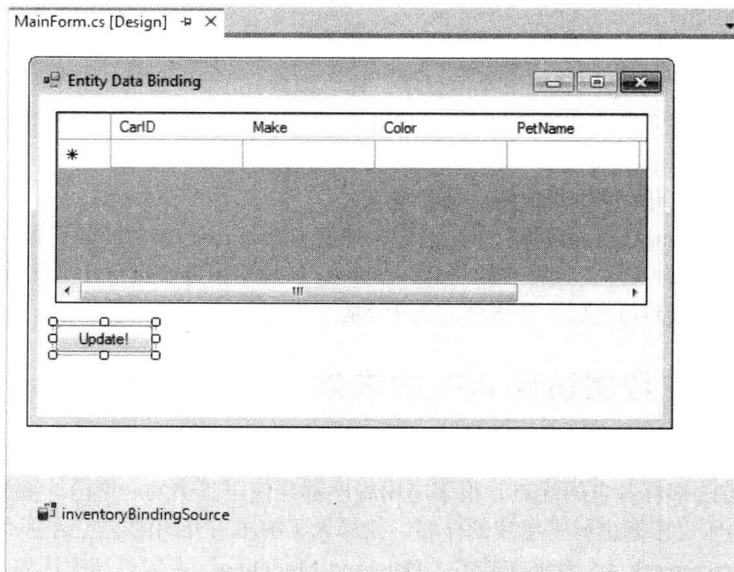


图23-25　最终UI

添加数据绑定代码

现在如果再添加一些代码，你的grid就可以显示任意数量的Inventory对象。代码再简单不过了，这要归功于EF的运行时引擎。为MainForm类的FormClosed、Load事件（使用Properties窗口）和Button控件的Click事件添加处理程序，代码片段如下：

```
public partial class MainForm : Form
{
  AutoLotEntities context = new AutoLotEntities();

  public MainForm()
  {
    InitializeComponent();
  }

  private void MainForm_Load(object sender, EventArgs e)
  {
    // 将ObjectSet<Inventory>集合绑定到grid
    gridInventory.DataSource = context.Inventories;
  }

  private void btnUpdate_Click(object sender, EventArgs e)
  {
    context.SaveChanges();
    MessageBox.Show("Data saved!");
  }

  private void MainForm_FormClosed(object sender, FormClosedEventArgs e)
  {
    context.Dispose();
  }
}
```

这就你需要做的全部工作！运行该应用程序，你可以向grid添加新的记录、选中一行然后删除，或修改已存在的行。对象上下文为查询、更新、插入自动生成了所有必要的SQL语句，因此单击Update按钮时，Inventory数据库将自动更新。以下是上例的几个关键之处。

- 上下文在整个应用程序中都保持已分配状态。
- 调用context.Inventories将执行SQL语句，把所有Inventory表中的数据提取到内存中。
- 上下文会跟踪脏的实体，这样在执行SaveChanges()时才知道要执行什么样的SQL语句。
- 执行完SaveChanges()之后，实体将变为干净的。

23.6 展望.NET 数据访问 API 的未来

在这三章中，我们介绍了三种使用ADO.NET操纵数据的方法，即连接层、断开连接层和Entity Framework。每种方法都有各自的优点，很多应用程序都会使用其中的一部分。要知道，我们只是对ADO.NET技术中的所有话题进行了粗浅的介绍。要深入了解本书介绍的这些话题（以及相关内容），我推荐参考.NET Framework 4.5 SDK文档的“Data and Modeling”主题。其中还包含了大量代码示例（如图23-26所示）。

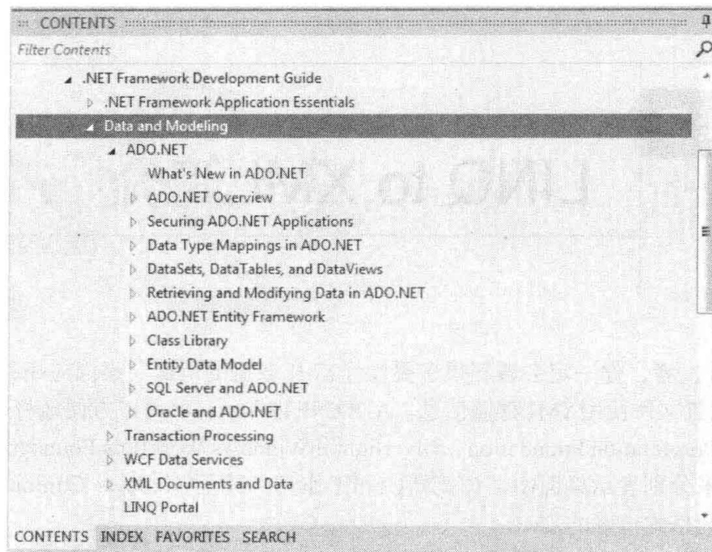


图23-26 .NET Framework文档Data and Modeling部分

23.7 小结

本章通过介绍Entity Framework的作用，结束了前面使用ADO.NET数据库编程的学习。EF允许你对更接近于业务领域的概念模型进行编程。你可以随意改造实体，而EF运行时将确保这些改变的数据被映射到正确的物理表数据。

你学习了*.edmx文件的作用（及其内容），以及如何使用Visual Studio IDE生成这些文件。同时，你还学习了如何将存储过程映射到概念层中的功能（function）、如何对物理模型使用LINQ查询、Entity SQL的作用，以及如何使用`EitityDataReader`最快捷地获取数据。

本章的最后描述了一个简单的示例，在Windows Forms应用程序环境中将实体类绑定到图形用户界面。在学习Windows Presentation Foundation和ASP.NET时你还将看到其他将实体绑定到GUI应用程序的示例。

23

第 24 章

LINQ to XML简介

作为.NET开发者，你一定会遇到很多要操作XML数据的情况。桌面应用程序和基于Web的应用程序配置文件使用XML存储信息。ADO.NET的DataSet可以方便地将数据保存（或加载）为XML。Windows Presentation Foundation、Silverlight和Windows Workflow Foundation都使用基于XML的语法（XAML）来分别表示桌面UI、浏览器UI和工作流。甚至Windows Communication Foundation也将很多设置保存为格式良好的XML。

尽管XML无处不在，但对一个不是很熟悉众多XML相关技术（XPath、XQuery、XSLT、DOM、SAX，等等）的人来说，对XML进行编程历来都是十分烦琐的。自.NET平台面世以来，微软就提供了一个特殊的程序集System.Xml.dll来专门对XML文档进行编程。这个二进制文件提供了大量命名空间和类型来应对不同的XML编程技术，和一些.NET特有的XML API，如XmlReader/XmlWriter类。

如今，大多数.NET程序员选择使用LINQ to XML API来与XML数据进行交互。正如你将在本章中所看到的那样，LINQ to XML编程模型允许你在代码中捕获XML数据的结构，并提供了极其简单的方式来创建、操作、加载和保存XML数据。你不仅可以使用LINQ to XML以非常简单的方式创建XML文档，而且还可以对LINQ查询表达式进行组合，来快速查询文档中的信息。

24.1 两个 XML API 的故事

当.NET平台面世的时候，程序员可以使用System.Xml.dll程序集中的类型来操作XML文档。使用该程序集下的命名空间和类型可以在内存中生成XML数据并保存到硬盘中。同样System.Xml.dll程序集还提供了一些类型用于将XML文件加载到内存、检索一个XML文档中的特定节点、验证文档是否符合某个架构（schema）以及其他常用编程任务。

尽管这个原始库成功地用于很多.NET项目中，但使用这些类型还是有一点点麻烦（说得客气点），因为编程模型与XML文档本身没有任何关联。例如，假设我们需要在内存中创建一个XML文件并将其保存到文件系统。如果使用System.Xml.dll中的类型，代码将会如下所示（创建一个名为LinqToXmlFirstLook的控制台应用程序项目，并添加System.Xml命名空间）：

```
private static void BuildXmlDocWithDOM()
{
  // 在内存中新建一个Xml文档
  XmlDocument doc = new XmlDocument();

  // 用根元素<Inventory>填充文档
  XmlElement inventory = doc.CreateElement("Inventory");
```

```
    // 现在创建一个<Car>子元素，它包含一个ID特性
    XmlElement car = doc.CreateElement("Car");
    car.SetAttribute("ID", "1000");

    // 创建<Car>元素中的数据
    XmlElement name = doc.CreateElement("PetName");
    name.InnerText = "Jimbo";
    XmlElement color = doc.CreateElement("Color");
    color.InnerText = "Red";
    XmlElement make = doc.CreateElement("Make");
    make.InnerText = "Ford";

    // 将<PetName>、<Color>、<Make>添加到<Car>元素
    car.AppendChild(name);
    car.AppendChild(color);
    car.AppendChild(make);

    // 将<Car>元素添加到<Inventory>元素
    inventory.AppendChild(car);

    // 将完整的XML插入到XmlDocument对象并保存文件
    doc.AppendChild(inventory);
    doc.Save("Inventory.xml");
}
```

在调用该方法时，你可以看到生成的Inventory.xml文件（位于\bin\Debug目录下），它包含的数据如下：

```
<Inventory>
  <Car ID="1000">
    <PetName>Jimbo</PetName>
    <Color>Red</Color>
    <Make>Ford</Make>
  </Car>
</Inventory>
```

虽然这种方法可以正常工作，但还是存在一些不足。首先，System.Xml.dll的编程模型是微软所实现的W3C文档对象模型（DOM）规范。在该模型下，XML文档是自下而上创建的。首先创建文档，然后创建子元素，最后将元素添加到文档中。在用代码实现它们时，你需要编写大量函数调用XmlDocument、XmlElement等类。

本例用了16行代码（不含注释）来创建这个极其简单的XML文档。如果使用System.Xml.dll程序集来创建更复杂的文档，最终的代码会更多。尽管可以使用各种循环或条件结构来创建节点从而使代码得到精简，但事实上代码体还是无法与最终的XML树产生任何视觉上的关联。

24.1.1 更优秀的DOM——LINQ to XML

LINQ to XML API是另一种创建、操作和查询XML文档的方式。它相比System.Xml的DOM模型使用了更多函数式的方法。你不必将单独的元素组合成XML文档，不必使用一组函数来更新XML树，你只需要编写如下所示的自上而下的代码：

```
private static void BuildXmlDocWithLINQToXml()
{
    // 使用更加函数式的方式创建XML文档
    XElement doc =
        new XElement("Inventory",
            new XElement("Car", new XAttribute("ID", "1000"),
                new XElement("PetName", "Jimbo"),
                new XElement("Color", "Red"),
                new XElement("Make", "Ford")
            )
        );

    // 保存到文件
    doc.Save("InventoryWithLINQ.xml");
}
```

这段代码使用了System.Xml.Linq命名空间下的一组新类型，如XElement和XAttribute。如果调用这段代码，你会发现它创建了与前面的例子相同的XML数据，但却少了很多不必要的麻烦。请注意，通过一些精心的缩进，代码与输出的XML文档的整体结构完全相同。这是非常有用的，而且它还比前面的例子少了很多代码（根据代码的放置情况，差不多节省了10行代码！）。

这段代码没有使用任何LINQ查询表达式，而只是使用了System.Xml.Linq命名空间下的一些类型来生成一个在内存中的XML文档，随后将其保存到文件中。我们将LINQ to XML视为一个更有效、更优秀的DOM。正如你将在本章后面所看到的，System.Xml.Linq中的类都是支持LINQ的，它们都可以作为同种类的LINQ查询的目标，正如第12章中所学到的那样。

随着对LINQ to XML学习的深入，你肯定会发现它比.NET中原来的XML库要易用得多。在新项目中使用System.Xml.dll的可能性被大幅降低，但这并不意味着你不会再使用原来库中的命名空间。

24.1.2　更优秀的LINQ to XML——VB字面量语法

在使用C#正式开始学习LINQ to XML之前，我想简单地提一句，Visual Basic语言对该API的函数式使用达到了更高的级别。在VB中我们可以使用XML字面量，它允许你在代码中直接将XElement指定为内联的XML标记流。假设存在一个VB项目，你可以创建下面的方法：

```
Public Class XmlLiteralExample
  Public Sub MakeXmlFileUsingLiterals()
    ' 注意，我们可以将XML数据内联到XElement中
    Dim doc As XElement = _
      <Inventory>
        <Car ID="1000">
          <PetName>Jimbo</PetName>
          <Color>Red</Color>
          <Make>Ford</Make>
        </Car>
      </Inventory>

    ' 保存至文件
    doc.Save("InventoryVBStyle.xml")
  End Sub
End Class
```

在VB编译器处理XML字面量时，会将XML数据映射为正确的LINQ to XML对象模型。实际上，在VB项目中使用LINQ to XML时，IDE已经知道了XML字面量语法仅仅是相关代码的速记符号。如图

24-1所示，在</Inventory>结束标记之后使用点操作符所得到的成员与在强类型的XElement之后使用点操作符得到的成员完全相同。

尽管本书是关于C#编程语言的，但几乎所有开发者都认为VB对XML的支持简直令人拍案叫绝。即使你是那种无法使用BASIC家族语言进行日常开发的程序员，我还是希望你能通过.NET Framework 4.5 SDK了解一下VB字面量语法。你会发现你能够将XML数据处理工作与专用的*.dll相分离，而这正是VB的功劳。



图24-1 VB XML字面量语法是LINQ to XML对象模型的速记符号

24.2 System.Xml.Linq 命名空间的成员

有些让人吃惊的是，LINQ to XML的核心程序集（System.Xml.Linq.dll）只在三个不同的命名空间下定义了很少的类型。这三个命名空间是System.Xml.Linq、System.Xml.Schema、System.Xml.xPath（如图24-2所示）。

核心的命名空间System.Xml.Linq包含了一组非常易于管理的类，它们代表一个XML文档的不同方面（元素、属性、XML命名空间、XML注释、处理指令等），如表24-1所示。

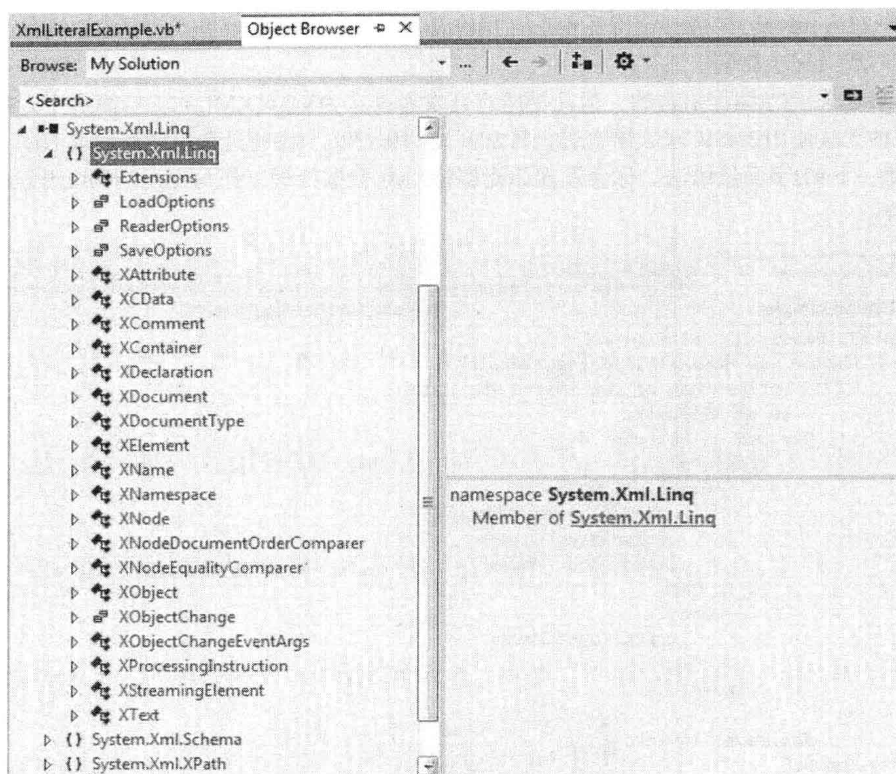


图24-2　System.Xml.Linq.dll中的命名空间

表24-1　选择System.Xml.Linq命名空间的成员

System.Xml.Linq的成员	含　义
XAttribute	表示一个XML元素的XML特性
XCData	表示XML文档中的CDATA部分。CDATA中的信息不必遵循XML的语法规则（如脚本代码）
XComment	表示一个XML注释
XDeclaration	表示一个XML文档中的公开声明
XDocument	表示一个XML文档的全部内容
XElement	表示一个XML文档中的特定元素，包含根元素
XName	表示一个XML元素或XML特性的名称
XNamespace	表示一个XML命名空间
XNode	表示XML树中节点（元素、注释、文件类型、处理指令或文本节点）的抽象概念
XProcessingInstruction	表示一个XML处理指令
XStreamingElement	表示一个支持延迟流输出的XML树

图24-3显示了这些关键类型的继承链。

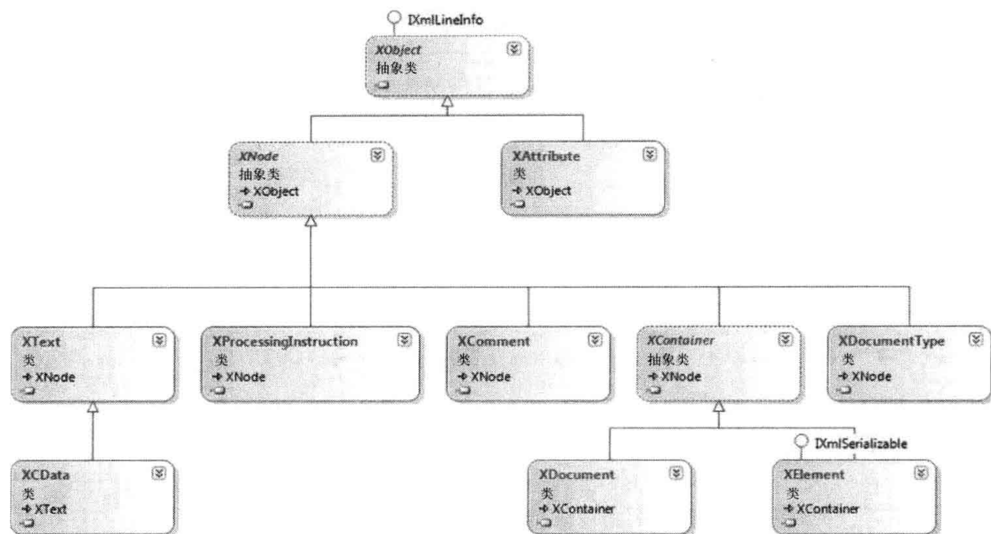


图24-3 LINQ to XML核心类的层次结构

24.2.1 LINQ to XML的轴方法

除了这些X*类，System.Xml.Linq中还定义了一个名为Extensions的类，它定义了一组针对IEnumerable<T>的扩展方法，其中T为XNode或XContainer的子类。表24-2列出了一些需要注意的重要的扩展方法（如你所见，这些方法在使用LINQ查询时是十分有用的）。

表24-2 LINQ to XML的Extensions类的Select成员

Extensions的成员	含　义
Ancestors<T>()	返回经过筛选的元素集合，其中包含源集合中每个节点的上级
Attributes()	返回源集合中经过筛选的每个元素的特性集合
DescendantNodes<T>()	返回集合中每个文档和元素的子代节点的集合
Descendants<T>	返回经过筛选的元素集合，其中包含源集合中每个元素和文档的子元素
Elements<T>	返回源集合中每个元素和文档的子元素的集合
Nodes<T>	返回源集合中每个文档和元素的子节点的集合
Remove()	将源集合中的每个特性从其父节点中移除
Remove<T>()	将源集合中出现的所有特定节点移除

顾名思义，这些方法允许在一个已加载的XML树中进行查询，以查找元素、特性以及他们的值。总的来说，这些方法被称为*轴方法*（axis method），或简称*轴*（axes）。这些方法可以直接用来处理节点树的一部分，也可以用来创建更复杂的LINQ查询。

说明　抽象的XContainer类支持很多与Extensions方法的名称相同的方法。由于XContainer是XElement和XDocument的父类，因此它们都支持全部的功能。

本章将介绍几个使用这些轴方法的示例，以下是一个快速示例：

```
private static void DeleteNodeFromDoc()
{
  XElement doc =
    new XElement("Inventory",
      new XElement("Car", new XAttribute("ID", "1000"),
        new XElement("PetName", "Jimbo"),
        new XElement("Color", "Red"),
        new XElement("Make", "Ford")
    )
  );

  // 从树中删除PetName元素
  doc.Descendants("PetName").Remove();
  Console.WriteLine(doc);
}
```

调用该方法后，你将看到如下所示的“修剪”后的XML树：

```
<Inventory>
  <Car ID="1000">
    <Color>Red</Color>
    <Make>Ford</Make>
  </Car>
</Inventory>
```

24.2.2　奇妙的XName和XNamespace

如果查看LINQ to XML轴方法（或XContainer中的同名成员）的签名，你会发现这些方法要求你指定一个XName对象。例如下面XContainer中定义的Desendants()方法的签名：

```
public IEnumerable<XElement> Descendants(XName name)
```

XName是很神奇的，因为你永远不需要在代码中直接使用它。事实上，由于该类没有公共构造函数，因此无法创建一个XName对象，如下所示：

```
// 错误！不能创建XName对象
doc.Descendants(new XName("PetName")).Remove();
```

如果查看XName的正式定义，你就会发现该类定义了一个自定义隐式转换操作符（参考第11章关于自定义转换操作符的定义），它会将一个简单的System.String映射到正确的XName对象：

```
  // 我们将在后台创建一个XName
doc.Descendants("PetName").Remove();
```

说明　XNamespace类同样支持隐式字符串转换。

这么做的好处是可以在使用这些轴方法时用文本值来表示元素或特性（attriubute）的名称，并允许LINQ to XML API将string数据映射到所需的对象类型。

源代码 LinqToXmlFirstLook示例的源代码位于Chapter 24子目录下。

24.3 使用 XElement 和 XDocument

让我们继续通过一个新的控制台应用程序ConstructingXmlDocs来研究LINQ to XML。在原始代码中引入System.Xml.Linq命名空间。正如已经介绍的那样，在LINQ to XML编程模型中，XDocument表示整个XML文档。它可以用来定义一个根元素及其包含的所有元素、处理指令和XML声明。下面是另一个使用XDocument构建XML数据的示例：

```
static void CreateFullXDocument()
{
  XDocument inventoryDoc =
    new XDocument(
      new XDeclaration("1.0", "utf-8", "yes"),
      new XComment("Current Inventory of cars!"),
      new XProcessingInstruction("xml-stylesheet",
        "href='MyStyles.css' title='Compact' type='text/css'"),
      new XElement("Inventory",
        new XElement("Car", new XAttribute("ID", "1"),
          new XElement("Color", "Green"),
          new XElement("Make", "BMW"),
          new XElement("PetName", "Stan")
        ),
        new XElement("Car", new XAttribute("ID", "2"),
          new XElement("Color", "Pink"),
          new XElement("Make", "Yugo"),
          new XElement("PetName", "Melvin")
        )
      )
    );

  // 保存到磁盘
  inventoryDoc.Save("SimpleInventory.xml");
}
```

请注意XDocument对象的构造函数实际上是其他LINQ to XML对象组成的树。这里所调用的构造函数的第一个参数为XDeclaration，然后是一个object型的参数数组（C#参数数组允许传递以逗号分隔的参数列表，它们将被打包成一个数组）：

```
public XDocument(System.Xml.Linq.XDeclaration declaration, params object[] content)
```

如果在Main()方法中调用该方法，将在SimpleInventory.xml文件中看到如下数据：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!--Current Inventory of cars!-->
<?xml-stylesheet href='MyStyles.css' title='Compact' type='text/css'?>
<Inventory>
  <Car ID="1">
```

```
    <Color>Green</Color>
    <Make>BMW</Make>
    <PetName>Stan</PetName>
  </Car>
  <Car ID="2">
    <Color>Pink</Color>
    <Make>Yugo</Make>
    <PetName>Melvin</PetName>
  </Car>
</Inventory>
```

事实上对于任何XDocument来说，默认的XML声明使用utf-8编码，XML版本为1.0，standalone特性为"yes"。因此，删除XDeclaration对象的创建将得到完全相同的数据。由于差不多每个文档都需要同样的声明，XDeclaration并不是很常用。

如果不需要定义处理指令或自定义XML声明，你可以不使用XDocument而简单地使用XElement。记住，XElement可以用来表示XML文档的根元素以及所有子对象。因此，可以像下面这样生成一个库存列表：

```
static void CreateRootAndChildren()
{
  XElement inventoryDoc =
    new XElement("Inventory",
    new XComment("Current Inventory of cars!"),
      new XElement("Car", new XAttribute("ID", "1"),
      new XElement("Color", "Green"),
      new XElement("Make", "BMW"),
      new XElement("PetName", "Stan")
    ),
    new XElement("Car", new XAttribute("ID", "2"),
      new XElement("Color", "Pink"),
      new XElement("Make", "Yugo"),
      new XElement("PetName", "Melvin")
      )
    );

    // 保存到磁盘
    inventoryDoc.Save("SimpleInventory.xml");
}
```

除了为一个不存在的样式表自定义了处理指令外，其余的输出结果基本一致：

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory>
  <!--Current Inventory of cars!-->
  <Car ID="1">
    <Color>Green</Color>
    <Make>BMW</Make>
    <PetName>Stan</PetName>
  </Car>
  <Car ID="2">
    <Color>Pink</Color>
    <Make>Yugo</Make>
    <PetName>Melvin</PetName>
  </Car>
</Inventory>
```

24.3.1 从数组和容器中生成文档

目前我们已经通过固定的硬编码方式构建了XML文档。而更常见的情况是通过从数组、ADO.NET对象、文件数据等诸如此类的数据源中读取数据来生成XElement（或XDocument）。使用一组标准的“for循环”来将数据移动到LINQ to XML对象模型，是一种将内存中的数据映射为新的XElement的方法。尽管这是可行的，但还是不如在LINQ查询中直接嵌入XElement的构造函数简便。

假设有一个匿名类的数组（这里只是为了精简代码，实际上可以为任何数组、List<T>和其他容器），可以用下面的代码将数据映射到XElement：

```
static void MakeXElementFromArray()
{
  // 创建一个匿名类型的数组
  var people = new[] {
    new { FirstName = "Mandy", Age = 32},
    new { FirstName = "Andrew", Age  = 40 },
    new { FirstName = "Dave", Age  = 41 },
    new { FirstName = "Sara", Age  = 31}
  };

  XElement peopleDoc =
    new XElement("People",
      from c in people select new XElement("Person", new XAttribute("Age", c.Age),
                              new XElement("FirstName", c.FirstName))
  );
  Console.WriteLine(peopleDoc);
}
```

这里的peopleDoc对象使用LINQ查询定义了根元素<People>。该LINQ查询根据people数组的每一项创建新的XElement。如果你觉得这种嵌入的查询有点不易阅读，那么完全可以像下面这样来断行：

```
static void MakeXElementFromArray()
{
  // 创建匿名类型的数组
  var people = new[] {
    new { FirstName = "Mandy", Age = 32},
    new { FirstName = "Andrew", Age  = 40 },
    new { FirstName = "Dave", Age  = 41 },
    new { FirstName = "Sara", Age  = 31}
  };

  var arrayDataAsXElements = from c in people
     select
       new XElement("Person",
         new XAttribute("Age", c.Age),
         new XElement("FirstName", c.FirstName));

  XElement peopleDoc = new XElement("People", arrayDataAsXElements);
  Console.WriteLine(peopleDoc);
}
```

总之，输出结果如下：

```
<People>
  <Person Age="32">
```

```
      <FirstName>Mandy</FirstName>
    </Person>
    <Person Age="40">
      <FirstName>Andrew</FirstName>
    </Person>
    <Person Age="41">
      <FirstName>Dave</FirstName>
    </Person>
    <Person Age="31">
      <FirstName>Sara</FirstName>
    </Person>
  </People>
```

24.3.2　加载和解析XML内容

XElement和XDocument都支持Load()和Parse()方法，可以从包含XML数据的string对象或外部XML文件中获取XML对象模型。考虑下面这段演示了这两个方法的代码：

```
static void ParseAndLoadExistingXml()
{
  // 从string中构建XElement
  string myElement =
    @"<Car ID ='3'>
      <Color>Yellow</Color>
      <Make>Yugo</Make>
    </Car>";
  XElement newElement = XElement.Parse(myElement);
  Console.WriteLine(newElement);
  Console.WriteLine();

  // 加载SimpleInventory.xml文件
  XDocument myDoc = XDocument.Load("SimpleInventory.xml");
  Console.WriteLine(myDoc);
}
```

源代码　ConstructingXmlDocs示例的源代码位于Chapter 24子目录下。

24.4　在内存中操作 XML 文档

到目前为止，我们介绍了使用LINQ to XML创建、保存、解析和加载XML数据的不同方法。接下来要关注的LINQ to XML方面是如何使用LINQ查询和LINQ to XML轴方法来导航一个指定的文档，对该文档进行定位并更改XML树中指定的项。

为此，我们需要创建一个Windows Form应用程序来显示保存在硬盘中的XML文档中的数据。GUI允许用户输入新节点的数据，并将其添加到同一个XML文档中。最后，将为用户提供一些方法，使用少量的LINQ查询在文档中执行搜索。

说明　由于已经在第12章中构建了很多LINQ查询，我不想在此再列举更多的查询了。如果你想看到更多的LINQ to XML示例，可以浏览.NET Framework 4.5 SDK文档中的“查询XML树”这个主题。

24.4.1　构建LINQ to XML应用程序的UI

创建一个名为LinqToXmlWinApp的Windows Form应用程序并将初始的Form1.cs文件名改为MainForm.cs（在Solution Explorer中）。该窗体的GUI非常简单。在窗体左边有一个TextBox控件（名为txtInventory），其Multiline属性为true，ScrollBars属性为Both。

除此之外，还有一组简单的TextBox控件（分别为txtMake、txtColor和txtPetName）和一个按钮btnAddNewItem，用户可以通过这些控件向XML文档中添加一条新的条目。最后，还有一组控件（一个名为txtMakeToLookUp的TextBox和一个名为btnLookUpColors的Button）用来查询XML文档中指定的节点。图24-4显示了界面的布局。

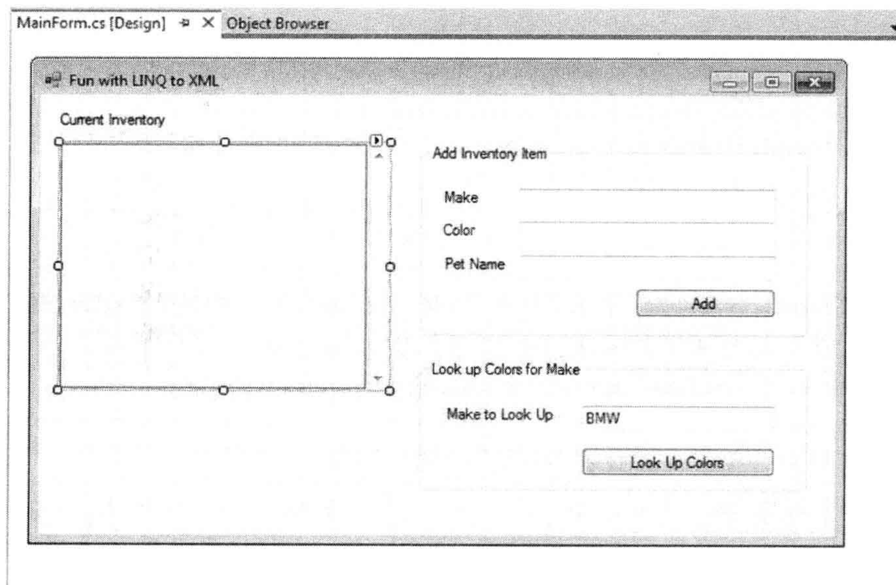


图24-4　LINQ to XML应用程序的GUI

为每个按钮的Click事件生成事件处理程序，同样还有窗体本身的Load事件。稍后我们将实现这些事件处理程序。

24.4.2　引入Inventory.xml文件

在本书的代码下载中该示例的解决方案代码里包含一个Inventory.xml文件。它的根元素<Inventory>下包含一些条目。选择Project→Add Existing Item菜单选项将该文件引入到项目中。当查看数据时你会发现在根元素下定义了一些<Car>元素，它们的定义与下面的类似：

```
<Car carID ="0">
  <Make>Ford</Make>
  <Color>Blue</Color>
  <PetName>Chuck</PetName>
</Car>
```

在继续下面的步骤之前，要确保在Solution Explorer中选择了该文件，然后使用Properties窗体将Copy to Output Directory属性设置为Copy Always。这可以保证在编译应用程序时在\bin\Debug文件夹下部署这些数据。

24.4.3　定义LINQ to XML辅助类

我们在LinqToXmlObjectModel项目中新建一个类，用来分离LINQ to XML数据。该类将定义一些静态方法来封装LINQ to XML逻辑。首先，定义一个方法，根据Inventory.xml文件的内容返回填充好的XDocument文档（需要在新建的文件中引入System.Xml.Linq和System.Windows.Forms命名空间）：

```
public static XDocument GetXmlInventory()
{
  try
  {
    XDocument inventoryDoc = XDocument.Load("Inventory.xml");
    return inventoryDoc;
  }
  catch (System.IO.FileNotFoundException ex)
  {
    MessageBox.Show(ex.Message);
    return null;
  }
}
```

InsertNewElement()方法（如下所示）获取“Add Inventory Item”中各个TextBox控件的值，并使用Descendants()轴方法将其放置到<Inventory>元素的新节点中。这些工作完成之后，将保存文档。

```
public static void InsertNewElement(string make, string color, string petName)
{
  // 加载当前文档
  XDocument inventoryDoc = XDocument.Load("Inventory.xml");

  // 为ID生成一个随机数
  Random r = new Random();

  // 根据传入参数新建XElement
  XElement newElement = new XElement("Car", new XAttribute("ID", r.Next(50000)),
    new XElement("Color", color),
    new XElement("Make", make),
    new XElement("PetName", petName));

  // 添加到内存中的对象
  inventoryDoc.Descendants("Inventory").First().Add(newElement);

  // 保存到磁盘
  inventoryDoc.Save("Inventory.xml");
}
```

最后一个方法LookUpColorsForMake()将使用一个LINQ查询获取最后一个TextBox中的数据，并创建一个包含指定的颜色字符串。考虑如下实现：

```
public static void LookUpColorsForMake(string make)
{
  // 加载当前文档
  XDocument inventoryDoc = XDocument.Load("Inventory.xml");

  // 根据给定的值查找颜色
  var makeInfo = from car in inventoryDoc.Descendants("Car")
        where (string)car.Element("Make") == make
        select car.Element("Color").Value;

  // 构建一个代表每个颜色的字符串
  string data = string.Empty;
  foreach (var item in makeInfo.Distinct())
  {
    data += string.Format("- {0}\n", item);
  }

  // 显示颜色
  MessageBox.Show(data, string.Format("{0} colors:", make));
}
```

24.4.4 将UI组装到辅助类

现在我们要做的就是实现事件处理程序的细节，只需要简单地调用静态的辅助方法，如下所示：

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
  }

  private void MainForm_Load(object sender, EventArgs e)
  {
    // 在TextBox控件中显示当前库存的XML文档
    txtInventory.Text = LinqToXmlObjectModel.GetXmlInventory().ToString();
  }

  private void btnAddNewItem_Click(object sender, EventArgs e)
  {
    // 为文档添加一个新项
    LinqToXmlObjectModel.InsertNewElement(txtMake.Text, txtColor.Text, txtPetName.Text);

    // 在TextBox控件中显示当前库存的XML文档
    txtInventory.Text = LinqToXmlObjectModel.GetXmlInventory().ToString();
  }

  private void btnLookUpColors_Click(object sender, EventArgs e)
  {
    LinqToXmlObjectModel.LookUpColorsForMake(txtMakeToLookUp.Text);
  }
}
```

图24-5显示了最终的输出结果。

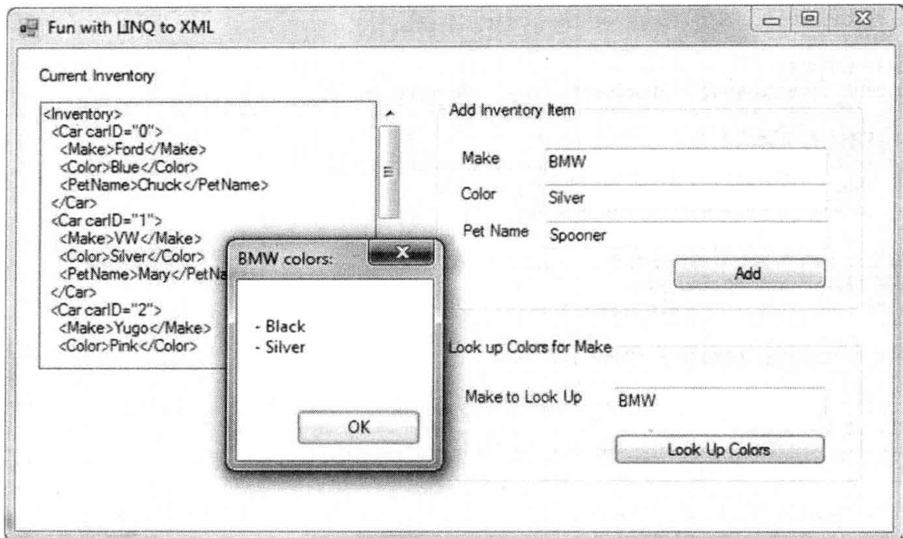


图24-5 完整的LINQ to XML应用程序

这就是所有对于LINQ to XML的介绍和关于LINQ的研究。我们首先在第12章引入了LINQ并学习了LINQ to Object。第19章演示了各种使用PLINQ的示例，而第23章展示了如何在ADO.NET Entity对象中使用LINQ。有了这些知识，你一定会精神饱满地继续深入研究。微软已经明确表示LINQ将会随着.NET平台的发展而继续不断改进。

源代码 XML示例LinqToXmlWinApp的源代码位于Chapter 24子目录下。

24.5 小结

本章考察了LINQ to XML的作用。如你所见，该API是随.NET平台发布的原始XML操作库System.Xml.dll的替代品。System.Xml.Linq.dll采用一种自上而下的方法生成新的XML文档，其代码结构与最终的XML数据极其接近。就此而论，LINQ to XML是更优秀的DOM。我们还学习了如何通过各种方式（解析、从文件加载、从内存中的对象映射）构建`XDocument`和`XElement`，以及如何使用LINQ查询导航和操作数据。

第 25 章 WCF

25

微软从.NET 3.0开始引入了一种专门用来构建分布式系统的API——WCF（Windows Communication Foundation）。与你过去所使用的其他分布式API（如DCOM、.NET Remoting、XML Web服务、消息队列等）有所不同，WCF提供了统一的、可扩展的编程对象模型来使用以前多个分布式技术。

本章首先通过快速浏览前面的分布式计算API来介绍为什么需要WCF并探究它所要解决的问题。在对WCF所提供的服务有了初步的了解之后，我们将转向探究代表WCF编程模型的.NET程序集、命名空间和类型。然后在本章的其余篇幅中，我们将使用不同的WCF开发工具构建几种WCF服务端、宿主和客户端。

说明 本章所包含的代码需要你以管理员权限启动Visual Studio（而且你也必须具有管理员权限）。要以正确的管理权限启动Visual Studio，可以右击Visual Studio图标，选择Run As Administrator。

25.1 各种分布式计算 API

Windows操作系统过去提供了许多用于构建分布式系统的API。虽然大多数人认为分布式系统包含至少两台联网的计算机，但是这个术语从广义上说只是指两个可执行程序需要交换数据，即使它们运行于相同的物理宿主。通过这个定义，为我们当前的编程任务选择分布式API往往需要解决如下关键问题：

这个系统仅在“内部”使用吗？还是外部用户也需要访问这个应用程序的功能？

如果我们构建内部使用的分布式系统，那么我们就有更多选择来确保每一个相连的计算机都运行于相同的操作系统，使用相同的编程模型（如.NET、COM或者Java平台）。运行内部系统也意味着可以使用既有的安全系统来处理授权、验证等。这样，我们就可能希望选择局限于特定操作系统/编程框架的特定分布式API来获得更好的性能。

相反，如果我们构建的系统必须从防火墙外部被访问，我们就要处理其他一些问题。首先，我们往往不能限制外部用户使用的操作系统、用来构建软件的编程框架或安全设置。

此外，如果是为使用许多操作系统和编程技术的大型公司或者大学工作，内部应用程序也就面临着和面向外部应用程序相同的挑战。对于这些情况，我们需要使用更灵活的分布式API来确保应用程序被尽可能地访问到。

基于针对这个关键的分布式计算问题的答案，下一个任务就是决定究竟要使用哪个API（或一组API）。在简单的概览之后，下面的小节会简单回顾Windows软件开发人员过去使用的一些主要分布式API，然后我们很容易就能看到WCF的有用之处了。

说明　为了确保我们说的是同一件事情，有必要指出WCF（以及它包含的技术）和构建基于HTML的网站没有什么关系。虽然Web应用程序被认为是“分布式的”，因为通常在交换中包含两个机器，但是WCF是用来创建对机器的连接以共享远程组件功能的——不是用于在Web浏览器上显示HTML。第32章会研究使用.NET平台来构建网站。

25.1.1　DCOM的作用

在.NET平台发布之前，DCOM（分布式组件对象模型）是微软相关开发人员可以选择的远程API。使用DCOM，我们可以使用COM对象、系统注册表（以及大量体力劳动）来构建分布式系统。DCOM的一个优势是运行组件的位置透明性。简单来说，可以不通过硬编码远程对象物理位置地址来为客户端软件编程。不管远程对象是在相同的机器上还是在二级网络计算机上，代码还是能保持不变，因为真实路径记录在外部的系统注册表中。

虽然DCOM确实有一定的成功，但是实际上它是Windows相关的API。即使很多其他操作系统支持DCOM，DCOM本身并没有提供一个结构用于构建包含多个操作系统（Windows、Unix、Max）的复杂解决方案或促进多个不同架构（COM、Java、CORBA等）的数据分享。

说明　虽然有很多尝试让DCOM能运行于各种形式的Unix/Linux上，但是最后的结果很无聊，并且最终变成了技术脚注。

总体来说，DCOM非常适用于内部的应用程序开发，因为把COM对象向公司防火墙外部公开会遇到很多额外的困难（防火墙等）。随着.NET平台的发布，DCOM马上就变成了遗留编程模型。如果我们不是在维护遗留的DCOM系统，就可以认为这项技术是不推荐的。

25.1.2　COM+/企业服务的作用

DCOM只是定义了一种在两个基于COM的软件之间创建通信通道的方式。为了完善用于构建功能丰富的分布式计算解决方案，微软最后发布了MTS（微软事务处理服务器），它在之后的发布中被命名为COM+。

不管名字是什么，不仅COM程序员能使用COM+，.NET人员也可以访问它。自从.NET平台第一次发布以来，基础类库就提供了一个叫`System.EnterpriseServices`的命名空间。在这里，.NET程序员可以构建能安装到COM+运行库的托管类库，这样就能像传统的COM+相关的COM服务器一样访问一组服务。在任何一种情况下，一旦COM+相关类库安装到了COM+运行库中，它就被称为服务组件。

COM+提供了许多服务组件可以利用的特性，包括事务管理、对象生命周期管理、池服务、基于

角色的安全系统、松耦合的事件模型等。这是COM+主要的优势，因为大多数分布式系统都需要相同的服务集合。COM+提供了现成的解决方案，而开发人员无需强制来手写代码。

COM+非常好的一个方面是所有这些配置都可以以声明方式使用管理工具进行配置。因此，如果希望确保对象受事务上下文监视或属于某个安全角色，只需要选择正确的复选框。

虽然COM+/企业服务至今仍在使用，但是这个技术只是Windows的解决方案，并且最适用于内部应用程序开发或作为被前端（如在后台调用服务组件COM+对象的公共网站）间接操作的后台服务。

说明 WCF当前没有提供构建服务组件的方式。然而，它却提供了一种方式让WCF服务和既有的COM+对象进行通信。如果你希望使用C#构建服务组件，我们需要直接使用`System.EnterpriseSerives`命名空间。细节请参考.NET Framework 4.5 SDK文档。

25.1.3 MSMQ的作用

MSMQ（微软消息队列）API可用来构建需要确保消息数据在网络上可靠传送的分布式系统。我们知道，任何分布式系统都存在网络服务器停机、数据库断开连接或由于各种原因引起的连接丢失等风险。此外，许多应用程序需要以这样一种形式来构建，那就是保存之后会传递的数据（叫做队列数据）。

首先，MSMQ被打包成为一组基于C的低级别API和COM对象。同样，使用`System.Messaging`命名空间，.NET程序员可以连接MSMQ并且以可靠方式构建和间断连接应用程序进行通信的软件。

另一方面，COM+层使用一种叫做队列组件（QC）的技术把MSMQ功能包含到了运行库中（以简化的形式）。这种与MSMQ通信的方式被打包到`System.EnterpriseServices`命名空间中。

不管我们使用哪种编程模型来和MSMQ运行库进行交互，最终结果确保了应用程序可以可靠并及时地传递数据。和COM+相似，MSMQ仍然是在Windows操作系统上构建分布式软件的结构的一部分。

25.1.4 .NET Remoting的作用

之前提过，随着.NET平台的发布，DCOM很快就成为了遗留分布式API。.NET基础类库附带了.NET Remoting层（用`System.Runtime.Remoting`命名空间表示）来替代其地位。如果所有应用程序都运行在.NET平台下，这个（遗留的）API就允许多个计算机来分布对象。

.NET Remoting API提供了许多有用的特性。最重要的就是使用基于XML的配置文件以声明方式定义客户端和服务端软件使用的基础通道。使用*.config文件，只需要改变配置文件的内容并且重启应用程序就可以简单有效地改变分布式系统的功能。

同样，由于这个API只能用于.NET应用程序，我们可以获得各种性能优势，因为数据可以以精简的二进制格式进行编码，并且在定义参数和返回值的时候可以使用公共类型系统（CTS）。虽然可以使用.NET Remoting来构建跨越多个操作系统的分布式系统（通过Mono，在第1章中简单介绍过），但是仍然不可以和其他编程架构（如Java）进行直接互操作。

25.1.5 XML Web服务的作用

前一节中每个分布式API都提供了一点（如果有的话）允许外部调用者以不可知的方式访问提供功能的支持。XML Web服务提供了最直接的方式，使我们可以将远程对象的服务向任何操作系统和任何编程模型公开。

和传统的基于浏览器的Web应用程序不同，Web服务只是通过标准Web协议公开远程组件功能的一种方式。自.NET最初发布以来，`System.Web.Services`命名空间就支持程序员构建和使用XML Web服务。其实，在很多情况下，构建功能完整的Web服务并不比在希望提供访问的公共方法上应用`[WebMethod]`特性复杂。此外，Visual Studio允许我们单击一两个按钮就可以连接到远程Web服务。

Web服务允许开发人员构建.NET程序集来包含通过简单HTTP就可以访问的类型。此外，Web服务以简单XML编码其数据。由于Web服务是基于工业开放标准的（HTTP、XML、SOAP等）而不是私有的类型系统和私有的有线格式（就像DCOM或.NET Remoting一样），它们允许高度的互操作性和数据交换。图25-1展示了XML Web服务的多变性。

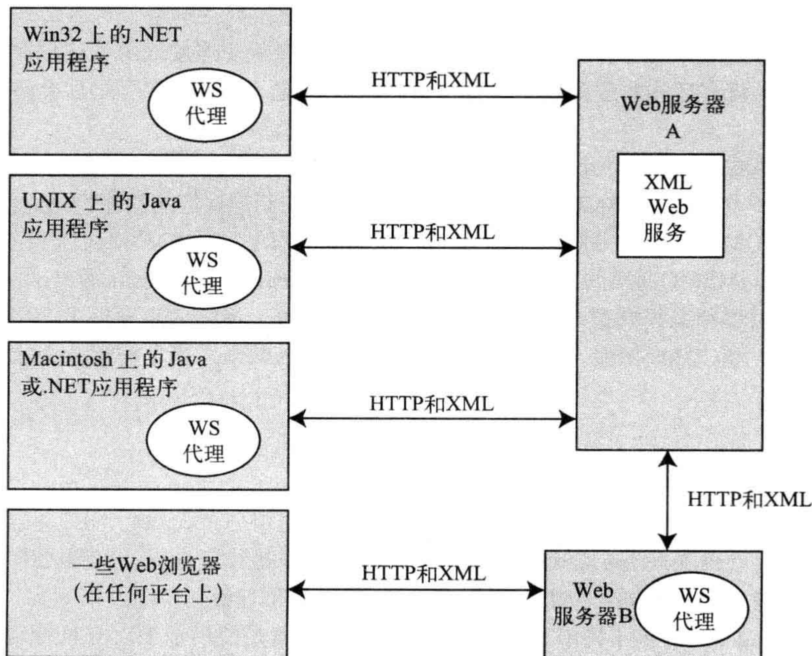


图25-1 XML Web服务具有非常高的互操作性

当然，没有一个分布式API是完美的。Web服务一个潜在的缺点是它们可能会有很多性能问题（因为HTTP和XML数据表示的使用），并且它们可能不是用于内部应用程序的理想解决方案，对内部应用程序来说，基于TCP的协议和二进制格式化的数据可能更适合。

Web服务标准

Web服务早期面临的另一个主要问题是，所有大型的行业公司（微软、IBM和SunMicroSystems）

创建的Web服务实现都不能和其他Web服务实现100%兼容。很明显，这是一个问题，因为Web服务的主要目的就是实现跨平台、跨操作系统的高度互操作性。

为了确保Web服务的互操作性，万维网联盟（W3C，www.w3.org）和Web服务互操作性组织（WS-I，www.ws-i.org）等已经开始编写规范来确保软件供应商（如微软、IBM以及SunMicroSystems）应该如何构建Web服务相关的软件类库来确保互操作性。

总体来说，所有这些规范都以WS-*作为名字并且覆盖了安全、附件以及Web服务描述（通过Web服务描述语言或WSDL）、策略、SOAP格式以及其他重要细节的问题。我们将看到，WCF支持很多WS-*规范。通常WCF服务会基于你选择的绑定来选择不同的WS-*规范。

说明 除了分布式API，开发人员还可以利用各种进程间通信协议，如命名管道和套接字。

25.2 WCF 的作用

从之前的一些内容中你已经发现，这么多分布式技术使得我们很难选择合适的。更复杂的是某些技术提供的功能还有重叠（最常见的就是事务和安全）。

即使.NET开发人员已经为手头的任务选择了看上去“正确”的技术，但是构建、维护和配置这样一个程序也很复杂。每一个API都有自己的编程模型、独有的配置工具等。因此，在WCF之前，如果不编写大量自定义基础结构的话，就很难实现即插即用的分布式API。例如，如果使用.NET Remoting API来构建系统，之后又觉得XML Web服务是更合适的解决方案，那么就需要重新写代码。

WCF是分布式计算工具包，它把之前这些独立的分布式技术整合到了主要由`System.ServiceModel`命名空间表示的简化API中。使用WCF，我们就可以使用大量的技术来将服务公开给调用者。例如，如果要构建一个内部的应用程序，所有计算机都是基于Windows的，就可以使用各种TCP协议来确保最快的性能。相同的服务也完全可以使用HTTP和SOAP进行公开，使得外部调用者可以利用其功能，而无须考虑编程语言和操作系统。

由于WCF允许我们为工作选择正确的协议（使用通用的编程模型），我们可以很容易地为分布式应用程序实现即插即用的基础管道。在大多数情况下，我们不需要重新编译或重新部署客户端/服务端软件就可以完成，因为这些细节都交给了应用程序配置文件。

25.2.1 WCF特性概览

不同API之间的互操作性和整合只是WCF中的两个重要方面。此外，WCF提供了一个丰富的软件框架，该框架与WCF所提供的远程技术相辅相成。请看下列WCF核心服务。

- 支持强类型与无类型消息。这就允许.NET应用程序共享自定义类型；与此同时，使用其他平台（如Java）创建的软件可以使用松散类型的XML流。
- 支持若干绑定（binding）（原始HTTP、TCP、MSMQ和命名管道），这就允许你选择最合适的方式去传输和接收数据。
- 支持最新和最高级的Web服务规范（WS-*）。

- 一个完全整合的安全模型，它包括原生Windows/.NET安全协议和许多构建在Web服务标准之上的中性安全技术。
- 对会话之类的状态管理技术的支持以及对单向无状态消息的支持。

虽然列出的这些特性用处很大，但它其实只是WCF的一小部分功能。此外，WCF还提供了跟踪和调试功能、性能计数器、发布订阅事件模型以及事务支持。

25.2.2　SOA概览

WCF的另外一个优势在于，它以SOA（*面向服务架构*）所确立的设计原则为基础。不可否认，SOA是业界的一个热点，和其他热点一样，SOA可以以各种方式来定义。简单来说，SOA是设计分布式系统的一种方式，它通过使用明确定义的*接口*通过跨越边界（可以是联网的计算机或只是同一个机器的两个进程）传递消息来让多个独立的*服务*协同工作。

在WCF的世界中，这些“明确定义的接口”通常使用实际的CLR接口类型来创建（见第9章）。然而，更广义地说，服务的接口只是描述了会由外部调用者调用的一组成员。

在设计WCF时，WCF团队贯彻了SOA设计准则的4个原则。虽然在构建WCF应用程序时，通常会自动遵守这些原则，但是理解SOA的这4个最主要的设计原则有助于我们更深刻地理解WCF。下面这部分内容简单介绍了每个原则。

1. 原则1：边界是明确的

这个原则强调的是WCF服务的功能是通过定义明确的接口进行表达的（如每一个成员的描述、其参数以及返回值）。外部调用者和WCF服务通信的唯一方式就是通过这个接口，并且外部调用者还是对底层的实现细节一无所知。

2. 原则2：服务是独立的

说服务是“独立”实体，我们指某个WCF服务（尽可能）是可以独立存在的。一个独立的服务对于版本问题、部署问题和安装问题来说应该是独立的。我们再来回顾一下基于接口编程的主要方面。在产品中接口应该永远不能修改（否则就会有打破既有客户端的危险）。如果我们需要为WCF服务扩展功能，只能编写新的接口来实现新的功能。

3. 原则3：服务通过契约而不是实现进行通信

第3个原则也是基于接口编程的另一个副产品，因为外部调用者不关心WCF服务的实现细节（用哪个语言写的，怎么完成工作的，等等）。WCF客户端仅仅通过它们公共的公共接口来和服务进行交互。此外，如果服务接口的成员公开了自定义的复杂类型，就需要完整描述*数据契约*的细节来确保所有调用者可以把内容映射到某个数据结构。

4. 原则4：服务兼容性基于策略

由于CLR接口为所有WCF客户端提供了强类型的契约（也会根据我们选择的绑定用来生成相关的WSDL文档），值得指出的是，接口/WSDL本身还不足够描述服务能够做什么的细节。因此，SOA允许我们定义*策略*来进一步限定服务的语法（如期望用于和服务通信的安全需求）。使用这些策略，我们就可以把服务如何工作和如何被调用的语法细节与服务低级别句法描述（公开的接口）进行分离了。

25.2.3 WCF概要

这些历史经验告诉我们WCF是构建分布式应用程序的首选方式。无论我们打算使用TCP协议构建内部的应用程序，使用命名管道在相同机器上转移数据，还是使用基于HTTP的协议向整个世界公开数据，WCF都是推荐的API。

这不是说有了新的开发方案，我们就不能使用原始的.NET分布式相关的命名空间了（System.Runtime.Remoting、System.Messaging、System.EnterpriseServices、System.Web.Services等）。其实，在一些情况下（特别是需要构建COM+对象时），我们也必须这么做。如果在之前项目中已经使用过这些API，你会发现学习WCF相当简单。和之前的技术相似，WCF使用了大量基于XML的配置文件、.NET特性和代理生成工具。

有了这样介绍性的内容之后，现在可以转到真实构建WCF应用程序的主题上来了。同样，对WCF的完整研究可能需要完整的一本书，因为支持的每一个服务（MSMQ、COM+、P2P、命名管道等）都可以写一章。在这里，我们会学习使用基于TCP和HTTP（如Web服务等）的协议构建WCF程序的完整步骤。这将为你将来的学习打下很好的基础。

25.3 WCF 核心程序集

正如你所想到的，WCF的功能由安装在GAC（全局程序集缓存）里面的一组.NET程序集所表示。表25-1描述了每一个WCF程序集的总体功能。

表25-1 WCF核心程序集

程 序 集	作 用
System.Runtime.Serialization.dll	该核心程序集定义了一些用于在WCF框架中序列化和反序列化对象的命名空间和类型
System.ServiceModel.dll	包含了核心类型的核心程序集。这些核心类型用于构建任何种类的WCF应用程序

表25-1列出的这两个核心程序集定义了一些新的命名空间和类型。表25-2列出了一些你应该知道的核心命名空间的作用，更加完整详细的说明请查阅.NET Framework 4.5 SDK文档。

表25-2 WCF核心命名空间

命名空间	作 用
System.Runtime.Serialization	定义了一些用来控制在WCF框架中如何序列化和反序列化数据的类型
System.ServiceModel	定义了绑定和承载类型，以及基础安全和事务类型
System.ServiceModel.Configuration	定义了提供对WCF配置文件核心部分进行编程访问的各种类型
System.ServiceModel.Description	定义了一些类型，为在WCF配置文件中定义的地址、绑定和契约提供对象模型
System.ServiceModel.MsmqIntegration	包含了用来与MSMQ服务进行整合的类型
System.ServiceModel.Security	定义了用来控制WCF安全层的种种类型

25.4 Visual Studio WCF 项目模板

WCF应用程序一般由三个相互关联的程序集表示（本章稍后会更详细解释），其中一个就是*.dll（换句话说就是WCF服务本身），外部调用者可以和它包含的类型进行通信。如果你希望构建WCF服务的话，完全可以选择标准的类库项目模板（见第14章）作为起始点并且手动引用WCF程序集。

或者还可以通过选择Visual Studio的WCF服务库（Service Library）项目模板来新建一个WCF服务，如图25-2所示。这个项目类型自动设置必要的WCF程序集引用，然而，它还生成很多常常会被删除的"初始代码"。

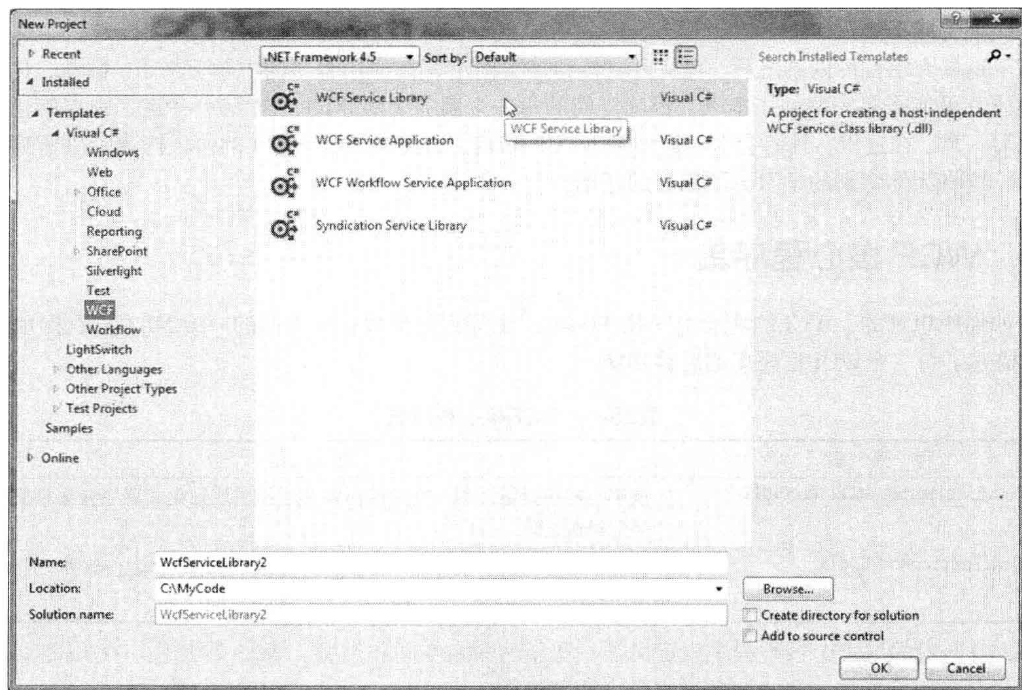


图25-2　Visual Studio WCF服务类库项目模板

选择WCF服务库项目模板的一个好处是，它还提供了一个App.config文件，可能你会感到奇怪，因为我们构建的是.NET *.dll而不是.NET *.exe。然而，这个文件在我们调试或者运行WCF服务库项目的时候非常有用，Visual Studio IDE会自动启动WCF测试客户端控制台。这个程序（WcfTestClient.exe）会读取App.config中的配置，为了测试目的而承载我们的服务。在本章后面，我们会学到更多有关WCF测试客户端的内容。

说明　WCF服务库项目的App.config文件同样有用，因为它显示了用于配置WCF宿主应用程序的基本设置。其实，你可以复制大多数代码并粘贴到产品服务的配置文件中。

除了基本WCF服务库模板外，New Project对话框的WCF项目分类定义了两个WCF库项目来将WF功能集成到WCF服务中，还定义了一个构建RSS库的模板（可以从图25-2中看到）。下一章会介绍WF，因此现在忽略这些特殊的WCF项目模板（我会把它留给感兴趣的读者来深入RSS提要项目模板）。

WCF服务网站项目模板

事实上，New Web Site对话框中，［通过File→New→WebSite菜单项来激活它（如图25-3所示）］还有另外一个Visual Studio WCF相关的项目模板。

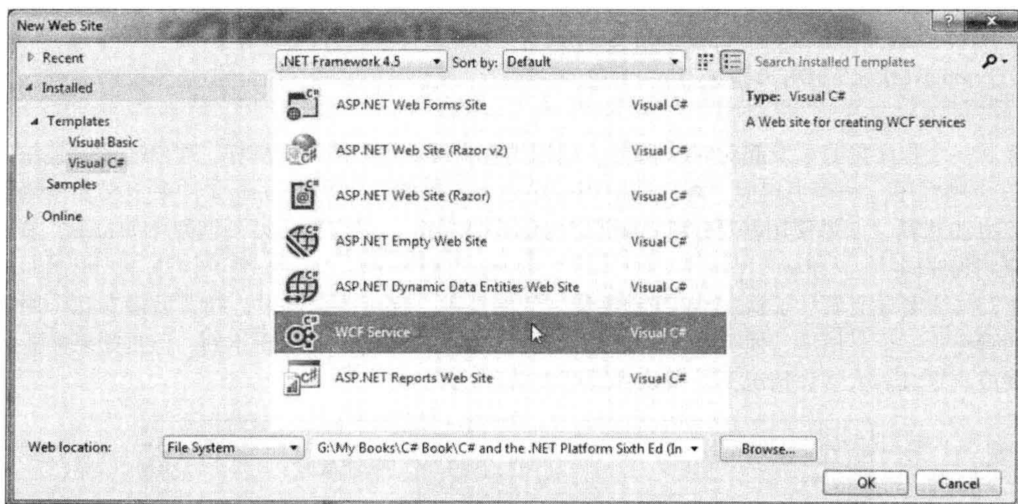


图25-3 Visual Studio基于Web的WCF服务项目模板

如果你一开始就知道WCF服务会使用基于HTTP的协议而不是诸如TCP或命名管道的协议，这个WCF服务项目模板很有用。这个选项可以自动创建新的IIS虚拟目录来包含我们的WCF程序文件，创建正确的web.config文件来通过HTTP公开服务并且编写必要的*.svc文件（本章后面会介绍更多有关*.svc文件的知识）。这样的话，这个基于Web的WCF服务项目只是节省了时间，因为IDE会自动设置必要的IIS基础结构。

相反，如果使用WCF服务库选项构建新的WCF服务的话，就可以以各种形式（自定义宿主、Windows服务、在IIS中手动创建虚拟目录等）来承载服务。如果你需要为WCF服务构建自定义宿主的话，这个选项就更合适。

25.5 WCF 应用程序的基本构成

构建WCF分布式系统时，一般会创建3个相互关联的程序集。

- **WCF服务程序集**：这个*.dll包含了表示希望向外部用户公开的整体功能的类和接口。
- **WCF服务宿主**：这个软件模块是承载WCF服务程序集的实体。
- **WCF客户端**：这是通过中间代理访问服务功能的应用程序。

之前说过，WCF服务程序集是一个.NET类库，它包含了许多WCF契约和它们的实现。唯一区别是接口契约使用各种特性修饰来控制数据类型表示、WCF运行库如何和公开的类型进行交互等。

第二个程序集WCF服务宿主可以是任何.NET可执行程序。在本章中我们会看到，WCF可以设置使用任何类型的应用程序（Windows Forms、Windows服务、WPF应用程序等）来公开。如果你在构建自定义宿主的话，就会使用`ServiceHost`类型和相关的*.config文件，它包含了有关希望使用的服务端管道的细节。然而，如果你使用IIS作为WCF服务宿主的话，就不需要以编程方式构建自定义宿主，因为IIS会在后台使用`ServiceHost`类型。

说明　还可以使用Windows Activation Service（WAS）来承载WCF服务，更多细节请参考.NET Framework 4.5 SDK文档。

最后一个程序集表示发起对WCF服务的调用的客户端。像你期望的那样，客户端可以是任何类型的.NET应用程序。和宿主相似，客户端应用程序通常还使用客户端*.config文件来定义客户端管道。你应该也注意到，如果使用基于HTTP的绑定构建WCF服务，很容易用另一框架（如Java）来编写客户端应用程序。

图25-4从宏观上显示了这3个WCF程序集之间的关系。你可能会假设，在背后还有几个用于表示必要的管道（工厂、信道、侦听器等）的底层细节。这些底层细节一般看不到，但是如果需要的话可以扩展或自定义。大多数情况下，默认管道可以满足要求。

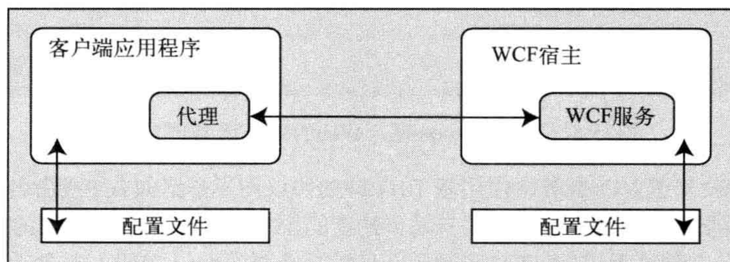


图 25-4　从宏观上看典型的 WCF 应用程序

还值得指出的是，从技术上说，使用服务端或客户端*.config文件是可选的。如果你希望的话，可以硬编码宿主和客户端来指定必要的管道（终结点、绑定、地址等）。使用这种方式明显的问题是，如果你需要改变管道细节，就需要重写代码，重新编译并且重新部署许多程序集。使用*.config文件能让我们的代码更灵活，因为改变管道就是更新文件内容并且重启应用程序这么简单。另一方面，在程序中编写的配置也可以使应用程序更灵活，例如，可以基于`if`条件测试来选择如何配置管道。

25.6　WCF 的 ABC

宿主和客户端相互通信会遵循ABC。再提醒一下，注意构建WCF应用程序的核心模块，具体而言就是地址（A，address）、绑定（B，binding）和契约（C，contract）。

- **地址**：服务的位置。在代码中，用`System.Uri`类型表示，然而，值一般保存在*.config文件中。

- **绑定**：WCF附带了许多不同的绑定来指定网络协议、编码机制和传输层。
- **契约**：从WCF服务公开的每一个方法的描述。

请务必了解，缩略语ABC并不意味着开发人员必须首先定义地址，然后定义绑定，最后定义契约。在许多情况下，WCF开发人员都是首先为服务定义契约，然后定义地址和绑定（其实它们的先后次序是不重要的，只要每个方面都考虑到即可）。在构建第一个WCF应用程序之前，先来研究这3个元素。

25.6.1 WCF契约

契约的概念对于构建WCF服务来说是关键的。尽管不是强制性的，但是绝大部分WCF应用程序都会首先定义一组.NET接口类型。这些类型用来表示某一给定的WCF类型将会支持的一组成员。确切地说，表示WCF契约的接口称作服务契约。实现了这些接口的类（或者结构）称作服务类型。

WCF服务契约标记了各种特性，最常见的特性定义在System.ServiceModel命名空间中。如果服务契约的成员只包含简单数据类型（如数值数据、布尔值和字符串数据），我们就可以只使用[ServiceContract]和[OperationContract]特性来构建一个完整的WCF服务。

然而，如果你的成员公开自定义类型，就会使用System.Runtime.Serialization.dll程序集中的System.Runtime.Serialization命名空间中（如图25-5所示）的不同类型。这里你可以用其他特性（如[DataMember]和[DataContract]）来完善定义过程，包括复合类型在传递给服务操作时如何序列化为XML，反之亦然。



图25-5 System.Runtime.Serialization定义了许多构建WCF数据契约时使用的特性

严格地说，我们不需要使用CLR接口来定义WCF契约。很多这些特性都可以应用在公共类（或结构）的公共成员上。然而，由于基于接口编程具有许多优势（多态、优雅的版本控制等），作为最佳实践还是应该考虑使用CLR接口来描述WCF契约。

25.6.2　WCF绑定

一旦定义并实现了一个（或者一组）契约后，接下来就该为WCF服务构建承载代理了。你会看到有多种承载方式可供选择，所有这些承载都必须指明一些绑定，远程调用者通过使用这些绑定来获取对服务类型功能的访问。

WCF附带了许多绑定选择，每一个都针定特定的需要。如果所有的现成绑定都不能满足要求，还可以通过扩展CustomBinding类型来创建自己的扩展（本章不会介绍这个）。简而言之，WCF绑定可以指定如下特性。

- 用于移动数据的传输层（HTTP、MSMQ、命名管道、TCP）。
- 用于传输的信道（单向、请求响应、双向）。
- 用于处理数据本身的编码机制（XML、二进制等）。
- 任何被支持的Web服务协议（如果绑定允许），如WS-Security、WS-Transaction、WS-Reliability等。

让我们看一下我们的选择。

1. 基于HTTP的绑定

BasicHttpBinding、WSHttpBinding、WSDualHttpBinding和WSFederationHttpBinding选项都可以通过HTTP/SOAP协议来公开契约类型。很明显，如果需要尽可能多地接触到服务（多操作系统和多编程架构），就可以关注这些绑定，因为所有这些类型都根据XML表示来编码数据并且在网络上使用HTTP。

表25-3中的每一种WCF绑定在代码中都可由System.ServiceModel命名空间中的一种类类型来表示，或者表示为*.config文件中所定义的XML属性。

表25-3　WCF提供的HTTP相关绑定

绑　定　类	绑定元素	含　　义
BasicHttpBinding	<basicHttpBinding>	用来构建符合WS-Basic Profile（WS-I Basic Profile 1.1）的WCF服务。该绑定使用HTTP作为传输，使用Text/XML作为默认的消息编码
WSHttpBinding	<wsHttpBinding>	和BasicHttpBinding相似，但是提供了更多Web服务特性。这个绑定增加了对事务、可靠消息和WS-Addressing的支持
WSDualHttpBinding	<wsDualHttpBinding>	和WSHttpBinding相似，但是用于双向契约（如服务和客户端可以互相传递消息）。这个绑定只支持SOAP安全并且需要可靠消息
WSFederationHttpBinding	<wsFederationHttpBinding>	一种支持WS-Federation协议，安全且可互操作的绑定。使联合（federation）内的组织可以高效地对用户进行验证和授权

正如其名，BasicHttpBinding是所有Web服务相关协议中最简单的。准确地说，这个绑定会确保我们的WCF服务遵守由WS-I定义的WS-I Basic Profile 1.1规范。使用这个绑定主要是为了维持之前和ASP.NET Web服务（从1.0版本开始就是.NET类库的一部分）通信的应用程序的向后兼容性。

WSHttpBinding协议不仅仅提供了对WS-*规范的支持（事务、安全和可靠会话），还支持使用消息传输优化机制（MTOM）处理二进制数据编码的能力。

WSDualHttpBinding的主要优势是它允许调用者和发送者使用双向消息进行通信。如果选择了WSDualHttpBinding，我们就可以挂接到WCF发布/订阅事件模型。

最后，如果安全性是最重要的话，可以考虑WSFederationHttpBinding这个基于Web服务的协议。这个绑定支持WS-Trust、WS-Security以及WS-SecureConversation规范，它们由WCF CardSpace API表示。

2. 基于TCP的绑定

如果你构建的分布式应用程序包括使用.NET 4.5类库进行配置的机器（换句话说，所有的机器都运行于Windows操作系统），我们就可以不使用Web服务绑定而使用TCP绑定来获取一些性能优势，它保证了所有数据都以紧凑二进制格式而不是XML来编码。同样，如果使用表25-4中列出的绑定，客户端和宿主必须是.NET应用程序。

表25-4　TCP相关的WCF绑定

绑　定　类	绑定元素	作　　用
NetNamedPipeBinding	<netNamedPipeBinding>	用于相同机器上.NET应用程序之间通信的安全的、可靠的、优化的绑定
NetPeerTcpBinding	<netPeerTcpBinding>	提供了P2P网络应用程序安全的绑定
NetTcpBinding	<netTcpBinding>	适合.NET应用程序跨机器通信的安全的优化的绑定

NetTcpBinding类使用TCP在客户端和WCF服务之间移动二进制数据。之前提过，这会比Web服务协议有更好的性能，但是这只能是内部的Windows解决方案。好的一面是NetTcpBinding支持事务、可靠会话和安全通信。

和NetTcpBinding相似，NetNamedPipeBinding支持事务、可靠会话和安全通信，但是不能跨机器调用。如果我们要以最快的方式在相同机器WCF应用程序之间（如跨应用程序域通信）推送数据，NetNamedPipeBinding是最佳选择。要了解NetPeerTcpBinding，请参考.NET Framework 4.5文档有关P2P网络的细节。

3. 基于MSMQ的绑定

最后，如果要整合微软MSMQ服务器，就要看一下NetMsmqBinding和MsmqIntergrationBinding绑定。在本章中，我们不会研究使用MSMQ绑定的细节，但是表25-5列举了每个绑定的基本作用。

表25-5　MSMQ相关的WCF绑定

绑　定　类	绑定元素	作　　用
MsmqIntegrationBinding	<msmqIntegrationBinding>	这个绑定可以用来让WCF应用程序向/从既有的使用COM、原生C++或定义在System.Messaging命名空间中的类型的MSMQ应用程序发送和接收消息
NetMsmqBinding	<netMsmqBinding>	这个绑定适用于跨机器通信的.NET应用程序。在MSMQ相关的绑定中，可以使用该绑定

25.6.3 WCF地址

创建契约和绑定之后，最后一个难题就是为WCF服务指定一个地址。很明显这很重要，因为远程调用者如果不能定位到远程类型的话，当然也就不能和它们进行通信。和WCF的很多方面一样，地址可以硬编码在程序集中（通过System.Uri类型）或分离到*.config文件。

对于任何一种情况，WCF地址确切的格式会根据我们选择的绑定类型有所不同（基于HTTP、命名管道、TCP或MSMQ）。从宏观上说，WCF地址可以指定如下一些信息。

- 构架：传输协议（HTTP等）。
- 机器名：机器的完全限定域名。
- 端口：在很多情况下是可选的。例如，HTTP绑定默认的端口是80。
- 路径：WCF服务的路径。

这个信息可以由如下泛化的模板来表示（端口值是可选的，因为很多绑定不会使用）：

```
构架://<机器名>[:端口]/路径
```

如果使用基于HTTP的绑定（basicHttpBinding、wsHttpBinding、wsDualHttpBinding或wsFederationHttpBinding），地址就可以是下面这样（回忆一下，如果没有指定端口号，基于HTTP协议的端口默认是80）：

```
http://localhost:8080/MyWCFService
```

如果你使用基于TCP的绑定（如NetTcpBinding或NetPeerTcpBinding），URI就应该是如下的格式：

```
net.tcp://localhost:8080/MyWCFService
```

MSMQ相关绑定（NetMsmqBinding和MsmqIntegrationBinding）对于URI格式有一些特殊，因为MSMQ可以使用公共的或私有（只在本机可用）的队列，并且端口号对于MSMQ相关的URI来说没意义。考虑如下URI，它描述了一个叫MyPrivateQ的私有队列：

```
net.msmq://localhost/private$/MyPrivateQ
```

最后，用于命名管道的绑定NetNamedPipeBinding地址格式可以为如下（回忆一下，命名管道允许同一个物理机器上的应用程序跨进程通信）：

```
net.pipe://localhost/MyWCFService
```

虽然一个WCF服务可能只会公开一个地址（基于一个绑定），但是还可以配置为一个地址的集合（使用不同的绑定）。可以通过在*.config文件中定义多个<endpoint>元素来实现。在这里，我们可以为相同的服务指定许多ABC。如果希望调用者在和服务通信的时候选择喜欢的协议，那么这个方式就很有用。

25.7 构建 WCF 服务

现在我们已经更好地理解了有关构建WCF应用程序的模块。让我们创建第一个示例应用程序来学习如何能在代码中表示ABC。第一个示例不会使用Visual Studio WCF项目模板，这样能更关注创建WCF服务中的一些细节步骤。

首先，新建一个叫MagicEightBallServiceLib的C#类库项目。完成后，把初始文件从Class1.cs重命名为MagicEightBallService.cs，然后添加对System.ServiceModel.dll程序集的引用。在初始代码文件中，指定我们要用的System.ServiceModel命名空间。至此，我们的C#代码应该差不多如下所示（注意这里的类是公共的）：

```
// 主要的WCF命名空间
using System.ServiceModel;

namespace MagicEightBallServiceLib
{
  public class MagicEightBallService
  {
  }
}
```

我们的类类型会实现由强类型CLR接口IEightBall表示的WCF服务契约。你可能知道，魔法8号球是一个玩具，能让我们看到你可能问的问题的一些固定回答。我们的接口会定义一个方法允许调用者对魔法8号球发起一个问题来获得随机答案。

WCF服务接口标记了[ServiceContract]特性，而每一个接口成员都标记了[OperationContract]特性（稍后会讲这两个特性的更多细节）。这里是IEightBall接口的定义：

```
[ServiceContract]
public interface IEightBall
{
  // 问一个问题，获得答案
  [OperationContract]
  string ObtainAnswerToQuestion(string userQuestion);
}
```

说明 还可以定义服务契约接口来包含没有使用[OperationContract]特性的方法。然而，这样的成员不会被WCF运行库公开。

从接口类型的学习（见第8章）中你可能知道了，接口在被类或结构实现（填充其功能）之前没什么用。和真实的魔法8号球一样，服务类型（MagicEightBallService）的实现会从字符串数组中随机返回一个答案。同样，我们的默认构造函数会显示一个在宿主控制台窗口中显示的消息（为了诊断目的）：

```
public class MagicEightBallService : IEightBall
{
  // 只是为了在宿主上显示
  public MagicEightBallService()
  {
    Console.WriteLine("The 8-Ball awaits your question...");
  }

  public string ObtainAnswerToQuestion(string userQuestion)
  {
```

```
    string[] answers =  { "Future Uncertain", "Yes", "No",
      "Hazy", "Ask again later", "Definitely" };

    // 返回随机的响应
    Random r = new Random();
    return answers[r.Next(answers.Length)];
  }
}
```

至此，我们的WCF服务库就完成了。在为服务构建宿主之前，先来研究[ServiceContract]和[OperationContract]特性的其他细节。

25.7.1　[ServiceContract]特性

为了使接口成为WCF所提供的服务，我们必须用[ServiceContract]特性来修饰它。和许多其他的.NET特性一样，ServiceContractAttribute类型支持许多用来限定其含义的属性。我们可以设置Name和Namespace这两个属性，用它们来控制服务类型的名称及其XML命名空间。当使用特定于HTTP的绑定时，这些值用来定义相关WSDL文档的<portType>元素。

由于服务类型的默认名称就是它在C#中的类名，所以我们在这里就不再浪费时间去给Name赋值了。然而，内部的XML命名空间的默认名称一般都是http://tempuri.org（对于WCF服务，你应该修改它）。

如果你构建的WCF服务会发送和接收自定义数据类型（现在我们没有这么做），为基础XML命名空间创建一个有意义的值就很重要，因为这样就可以让我们的自定义类型是唯一的。如果你有构建XML Web服务经验的话，就应该知道，XML命名空间提供了一种方式以唯一的容器包装我们的自定义类型，从而确保我们的类型不会和其他赋值的类型发生冲突。

因此，我们可以使用更合适的定义来更新我们的接口定义，这和在.NET Web服务项目中定义XML命名空间的过程很相似，一般设置为服务起始点的URL，例如：

```
[ServiceContract(Namespace = "http://MyCompany.com")]
public interface IEightBall
{
  ...
}
```

除了Namespace和Name之外，我们可以用表25-6中的另外一些属性来配置[ServiceContract]特性。根据你所选择的绑定，其中有一些设置可能会被忽略。

表25-6　[ServiceContract]特性的一些属性

属　性	含　义
CallbackContract	对于双向信息交换，如果服务契约需要具有回调功能，则要设置此属性
ConfigurationName	该名称用来在应用程序配置文件中定位服务元素。默认为实现服务的类的名称
ProtectionLevel	允许你为公开该契约的终结点指定契约绑定所需的安全级别，是要加密，需数字签名，或既要加密又需数字签名
SessionMode	允许你设定服务契约是支持会话，不支持会话，还是必须使用会话

25.7.2 [OperationContract]特性

在WCF框架中使用的方法必须用[OperationContract]特性来修饰。我们也可以用各种属性来配置该特性。使用表25-7所示的属性，你可以将一个给定的方法声明为在本质上是单向的，支持异步调用，需要加密的消息数据等（同样，根据所选择的绑定，许多值可能会被忽略）。

表25-7 [OperationContract]特性的属性

属　性	含　义
AsyncPattern	指明操作是否使用Begin/End方法对来异步执行。这能让服务把处理转到另一个服务器端线程上，这和客户端异步调用方法没有关系
IsInitiating	指定该操作能否作为一个会话中的初始操作
IsOneWay	指明该操作是否只是由单个输入消息组成（而并没有相关的输出）
IsTerminating	指定WCF运行库在操作完成后是否应该尝试去结束当前会话

对于第一个例子，我们并不需要用额外的特征来配置ObtainAnswerToQuestion()方法。因此，可以使用当前定义的[OperationContract]特性。

25.7.3 作为操作契约的服务类型

构建WCF服务类型并不需要使用接口。实际上，直接将[ServiceContract]特性和[OperationContract]特性应用到服务类型上也是可行的，如下所示。

```
// 这段代码只是为了演示，并不会用于当前示例
[ServiceContract(Namespace = "http://MyCompany.com")]
public class ServiceTypeAsContract
{
  [OperationContract]
  void SomeMethod() { }

  [OperationContract]
  void AnotherMethod() { }
}
```

虽然这种方法是可行的，但是显式定义一个接口类型去表示服务契约有很多好处。最明显的一个好处是给定的接口可被应用到多个服务类型（这些服务类型是用多种语言和架构来编写的），以此来获取高度的多态性。另外一个好处是一个服务契约接口可以用作新契约的基础（通过继承接口），而并不需要浪费时间去实现它。

到目前为止，第一个WCF服务库就完成了。编译项目以确保没有录入错误。

源代码　MagicEightBallServiceLib项目的源代码位于Chapter 25子目录MagicEightBallServiceHTTP下。

25.8　承载 WCF 服务

现在，我们已经为定义我们的服务类型的宿主做好了准备。虽然产品级的服务最好是由Windows服务或IIS虚拟目录承载，但是我们将创建一个名为MagicEightBallServiceHost的基于控制台的宿主。

创建完该项目以后，添加对System.ServiceModel.dll和MagicEightBallServiceLib.dll的引用，并且使用System.ServiceModel和MagicEightBallServiceLib命名空间来修改初始代码文件，如下所示：

```
using System;
...

using System.ServiceModel;
using MagicEightBallServiceLib;

namespace MagicEightBallServiceHost
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Console Based WCF Host *****");
      Console.ReadLine();
    }
  }
}
```

当构建WCF服务类型的宿主时，第一件必须做的事情是，你要决定是完全在代码中定义必要的承载逻辑，还是将一些低级的细节转移到应用程序配置文件中去。回想一下，*.config文件的好处是宿主能够改变底层的运行方式而并不需要重新编译和部署可执行文件。然而，请记住，这确实只是选择性的，因为你能够使用System.ServiceModel.dll程序集中的类型来对承载逻辑进行编码实现。

这个基于控制台的宿主确实会使用应用程序配置文件，因此使用Project→Add New Item菜单项，然后选择Application Configuration File，向当前项目插入一个新的应用程序配置文件（如果项目目前还没有配置文件的话）。

25.8.1　在App.config文件中创建ABC

在构建WCF服务类型的宿主时，你将会遵循一些常见的步骤（一部分是通过配置，一部分是通过代码），如下所示。

- 在宿主的配置文件中，定义所承载的WCF服务的终结点。
- 通过编程使用ServiceHost类型去提供终结点所提供的服务类型。
- 确保宿主保持运行状态，以处理所收到的客户端请求。显然，如果你使用Windows服务或IIS来承载服务类型，那么并不需要这一步。

在WCF的世界中，术语*终结点*只是表示地址、绑定和契约的一个包装。在XML中，终结点使用<endpoint>元素和address、binding以及contract元素进行表达。更新我们的*.config文件来指定一个由宿主公开的终结点（通过8080端口可用），如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="MagicEightBallServiceLib.MagicEightBallService">
        <endpoint address ="http://localhost:8080/MagicEightBallService"
                  binding="basicHttpBinding"
                  contract="MagicEightBallServiceLib.IEightBall"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

注意<system.serviceModel>元素是所有宿主WCF设置的根。每一个由宿主公开的服务都由<service>元素表示，并包装在<services>根元素中。在这里，一个<sercice>元素使用（可选的）name特性来指定服务类型的友好名称。

嵌套的<endpoint>元素处理定义地址、绑定模型（在本例中是basicHttpBinding）以及定义WCF服务契约接口的完全限定名（IEightBall）的任务。因为我们使用的是基于HTTP的绑定，所以可以使用https://架构随便指定一个端口ID。

25.8.2 针对ServiceHost类型进行编程

有了现在这样的配置文件，完成宿主的编程逻辑就很简单了。在启动可执行程序时，我们会创建ServiceHost类的实例并且通知它宿主的是哪个WCF服务。在运行时，这个对象会自动读取宿主*.config文件中<system.serviceModel>元素中的数据来检测正确的地址、绑定类型和契约。然后创建必要的管道：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Console Based WCF Host *****");

  using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
  {
    // 打开宿主并且开启对传入消息的监听
    serviceHost.Open();

    // 使服务保持运行状态，直到Enter键被按下
    Console.WriteLine("The service is ready.");
    Console.WriteLine("Press the Enter key to terminate service.");
    Console.ReadLine();
  }
}
```

运行这个应用程序，就会发现，宿主在内存中随时准备接收从远程客户端发来的请求。

说明 记住必须以管理员权限启动Visual Studio才能运行多种WCF项目类型！

25.8.3 指定库地址

现在，我们使用只需要服务类型信息的构造函数来创建ServiceHost。然而，还可以传入System.Uri

类型的数组作为构造函数参数来表示可以访问服务的地址集合。现在，地址是从*.config文件中获取的，然而如果按如下所示更新using区域的话：

```
using (ServiceHost serviceHost = new
  ServiceHost(typeof(MagicEightBallService),
  new Uri[]{new Uri("http://localhost:8080/MagicEightBallService")}))
{
  ...
}
```

我们就可以这样定义终结点：

```
<endpoint address =""
          binding="basicHttpBinding"
          contract="MagicEightBallServiceLib.IEightBall"/>
```

当然，在宿主代码库中有太多硬编码会减少灵活性。因此对于当前宿主的示例，假设你只是像前面那样使用类型信息来创建服务宿主：

```
using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
{
  ...
}
```

编写*.config文件时，根据在代码库中硬编码的量（就像刚才例子中看到的可选的Uri数组），我们有许多方式来构建XML的描述。为了演示另外一种编写*.config文件的方式，考虑如下的改造工作：

```
<?xml version= "1.0" encoding= "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name= "MagicEightBallServiceLib.MagicEightBallService">

        <!-- 从<baseAddresses>获取的地址 -->
        <endpoint address = ""
                  binding= "basicHttpBinding"
                  contract= "MagicEightBallServiceLib.IEightBall"/>

        <!-- 在专门的地方列出所有的根地址 -->
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

在这里，<endpoint>元素的address特性还是空的，无论我们在创建ServiceHost时，在代码中是否指定了Uri对象的数组，应用程序还是可以像以前那样运行，因为值可以从baseAddresses中得到。把根地址保存在<host>的<baseAddresses>区域中的一个优势是，*.config文件另外的一部分也需要知道服务终结点的地址。因此，我们可以这样把这个值进行隔离，而无须在一个*.config文件中复制并粘贴地址值。

说明　在稍后的示例中，我会介绍允许我们以更有趣的方式编写配置文件的图形配置工具。

不管怎么样，在构建客户端应用程序和服务进行通信之前，让我们深入ServiceHost类类型、<service.serviceModel>元素和元数据交换（MEX）服务的作用。

25.8.4 ServiceHost类型的功能

ServiceHost类型用于从承载可执行文件中配置和提供一种WCF服务。请注意，只有当构建一个自定义的*.exe来承载服务时，你才会直接使用该类型。如果你正在使用IIS来提供某个服务，它们会自动为你创建ServiceHost对象。

如你所看到的，ServiceHost类型需要完整的服务描述，该描述从宿主的*.config文件的配置设置动态获取。虽然对描述的获取是在对象创建期间自动发生的，但是你可以使用许多成员去手动配置ServiceHost对象的状态。除了以同步方式和你的服务进行通信的Open()和Close()外，表25-8还列出了其他一些有趣的成员。

表25-8　ServiceHost类型中的一部分成员

成　员	含　义
Authorization	该属性获取被承载服务的授权级别
AddDefaultEndpoints()	该方法使用框架提供的预置终结点，以编程方法配置WCF服务宿主
AddServiceEndpoint()	该方法允许你通过编程的方式在宿主中注册一个终结点
BaseAddresses	该属性获取已经在当前服务中注册的基本地址列表
BeginOpen()和BeginClose()	这两个方法允许你使用标准的异步.NET委托句法去异步打开和关闭一个ServiceHost对象
CloseTimeout	该属性允许你设置和获取服务关闭的允许时间
Credentials	该属性获取当前服务所使用的安全凭证
EndOpen()和EndClose()	这两个方法与BeginOpen()和BeginClose()是相对应的
OpenTimeout	该属性允许你设置和获取服务启动的允许时间
State	该属性获取一个显示通信对象当前状态的CommunicationState枚举值（opened、closed、created等）

为了演示ServiceHost的其他方面，使用新的静态方法更新我们的Program类，来输出当前宿主的各个方面：

```
static void DisplayHostInfo(ServiceHost host)
{
  Console.WriteLine();
  Console.WriteLine("***** Host Info *****");

  foreach (System.ServiceModel.Description.ServiceEndpoint se
    in host.Description.Endpoints)
  {
    Console.WriteLine("Address: {0}", se.Address);
    Console.WriteLine("Binding: {0}", se.Binding.Name);
    Console.WriteLine("Contract: {0}", se.Contract.Name);
    Console.WriteLine();
  }
  Console.WriteLine("**********************");
}
```

假设我们打开宿主后在Main()方法中调用这个新的方法：

```
using (ServiceHost serviceHost = new ServiceHost(typeof(MagicEightBallService)))
{
  // 打开宿主并且启动对传入消息的监听
  serviceHost.Open();
  DisplayHostInfo(serviceHost);
...
}
```

将会看到如下所示的统计信息：

```
***** Console Based WCF Host *****

***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
**********************
The service is ready.
Press the Enter key to terminate service.
```

25.8.5　<system.serviceModel>元素的细节

和任何XML元素一样，<system.serviceModel>元素可以包含许多子元素，而每一种子元素又可以用各种不同的特性来完整配置。.NET Framework 4.5 SDK文档描述了每一种特性的完整细节，我们在这里列出了<system.ServiceModel>元素部分有用的子元素。

```
<system.serviceModel>
  <behaviors>
  </behaviors>
  <client>
  </client>
  <commonBehaviors>
  </commonBehaviors>
  <diagnostics>
  </diagnostics>
  <comContracts>
  </comContracts>
  <services>
  </services>
  <bindings>
  </bindings>
</system.serviceModel>
```

到本章后面，会有更有趣的配置文件，表25-9列出了每个子元素的主要作用。

表25-9　<service.serviceModel>的部分子元素

子　元　素	作　　用
behaviors	WCF提供了各种终结点和服务行为。简而言之，行为允许我们进一步限定宿主、服务或客户端的功能
bindings	这个元素允许我们微调每个WCF提供的绑定（basicHttpBinding和netMsmqBinding等）和指定宿主使用的客户端绑定

（续）

子 元 素	作　用
client	这个元素包含客户端用来连接到服务的终结点列表。显然，这对于宿主的*.config文件没什么用
comContracts	这个元素定义了允许WCF和COM互操作的COM契约
commonBehaviors	这个元素只能在machine.config文件中进行设置。它可以用于定义某个机器所有WCF服务的行为
diagnostics	这个元素包含WCF诊断特性的设置。用户可以启用或禁用跟踪、性能计数器和WMI提供程序，并且可以增加自定义消息过滤器
services	这个元素包含了从宿主公开的WCF服务的集合

25.8.6　启用元数据交换

回忆一下，WCF客户端应用程序通过中间代理类型和WCF服务通信。虽然我们可以完全手写编写代理代码，但这样做很乏味而且会导致错误。最好能有一个工具可以生成必要的代码（包括客户端的*.config文件）。幸好.NET Framework 4.5 SDK提供了一个命令行工具（svcutil.exe）来实现。同样，Visual Studio通过Project→Add Service Reference菜单项提供了相似的功能。

然而，为了让这些工具生成必要的代理代码和*.config文件，它们必须能发现WCF服务接口的格式和任何已定义的数据契约（方法名、参数类型等）。

元数据交换（MEX）是一个WCF服务行为，可以指定它微调WCF运行库如何处理我们的服务。简而言之，每一个<behavior>元素都可以定义某个服务可以订阅的一组活动。WCF提供了许多现成的行为，我们还可以构建自己的。

MEX行为（默认是关闭的）会通过HTTP GET拦截任何元数据请求。如果你希望允许svcutil.exe或Visual Studio来自动创建必要的客户端代理*.config文件，就必须启用MEX。

只需要稍稍改动宿主的*.config文件，加上正确的设置（或编写相应的C#代码），就可以启用MEX了。首先，必须为MEX增加一个新的<endpoint>。其次，我们需要定义一个WCF行为来允许HTTP GET访问。再次，我们需要通过<service>元素的behaviorConfiguration特性来关联服务和行为。最后，我们需要增加一个<host>元素来定义这个服务的根地址（MEX会查看这里以找到要描述类型的位置）。

说明　如果你把表示根地址的System.Uri对象作为ServiceHost构造函数参数传入的话，就可以忽略最后一步。

考虑如下更新后的宿主*.config文件，它创建了自定义的<behavior>元素（命名为EightBall-ServiceMEXBehavior），并通过<service>定义中的behaviorConfiguration特性关联到我们的服务：

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService"
               behaviorConfiguration = "EightBallServiceMEXBehavior">
        <endpoint address = ""
```

```
                binding = "basicHttpBinding"
                contract = "MagicEightBallServiceLib.IEightBall"/>

        <!-- 启用MEX终结点 -->
        <endpoint address = "mex"
                binding = "mexHttpBinding"
                contract = "IMetadataExchange" />

        <!-- 需要增加这个，让MEX知道服务的地址 -->
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>

    <!-- MEX的行为定义 -->
    <behaviors>
      <serviceBehaviors>
        <behavior name = "EightBallServiceMEXBehavior" >
          <serviceMetadata httpGetEnabled = "true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

现在我们就可以重启服务宿主应用程序并且使用喜欢的Web浏览器来看元数据描述。在宿主运行的时候输入如下URL作为地址：

```
http://localhost:8080/MagicEightBallService
```

在WCF服务的主页上（如图25-6所示），我们得到了有关如何以编程方式和这个服务进行交互的基本细节，并且可以通过单击页面顶部的链接来查看WSDL契约。回忆一下，WSDL（Web服务描述语言）是描述某个终结点Web服务的语法。

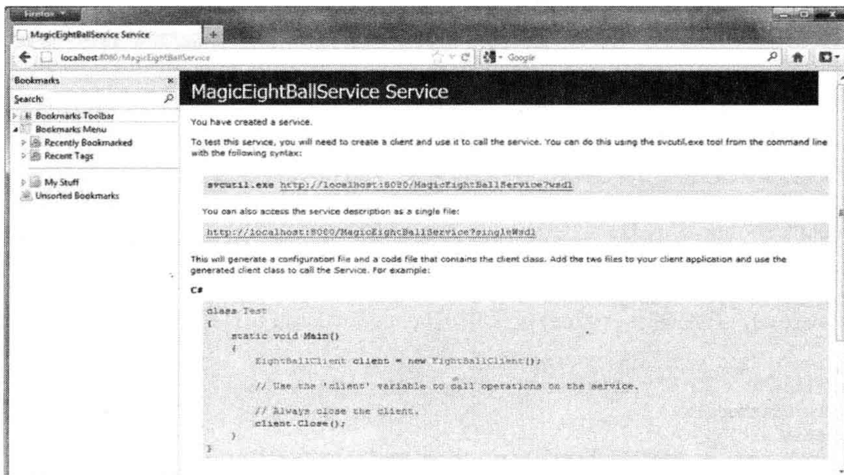


图25-6　准备通过MEX查看元数据

现在宿主公开两个不同的终结点（一个用于服务，一个用于MEX），宿主的控制台输出类似下面所示：

```
***** Console Based WCF Host *****

***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall

Address: http://localhost:8080/MagicEightBallService/mex
Binding: MetadataExchangeHttpBinding
Contract: IMetadataExchange
**********************

The service is ready.
```

源代码 MagicEightBallServiceHost项目的源代码位于Chapter 25的MagicEightBallServiceHTTP子目录下。

25.9 构建 WCF 客户端应用程序

现在，我们的宿主已经建立完毕。最后的任务是构建与该WCF服务类型进行通信的程序。虽然我们可以绕弯路去手动构建必要的基础结构（一种可行但很费劲的做法），但是.NET Framework 4.5 SDK已经为我们提供了一些构建客户端代理的方法。首先创建一个新的控制台应用程序项目MagicEightBall ServiceClient。

25.9.1 使用svcutil.exe生成代理代码

第一种创建客户端代理的方式是使用svcutil.exe命令行工具。可以使用该工具生成一个充当代理代码的C#语言文件以及一个客户端配置文件。为了生成这两个文件，只需要指定服务的终结点作为第一个参数。/out:标记用来定义包含代理的*.cs文件的名称，而/config:选项则指明所生成的客户端*.config文件的名称。

假设服务正在运行，那么传入svcutil.exe的下列命令将会在工作目录中生成两个新文件（当然，在Developer Command Prompt中输入时，这个命令只有一行）：

```
svcutil http://localhost:8080/MagicEightBallService
        /out:myProxy.cs /config:app.config
```

如果你打开myProxy.cs文件，会看见IEightBall接口的客户端表示，以及一个名为EightBallClient的新类，这个类就是代理本身。EightBallClient类派生自System.ServiceModel.ClientBase<T>泛型类（在这里，T是已注册的服务接口）。除了一些生成的构造函数之外，该代理的每个方法（基于原始的接口方法）还实现了所有用[OperationContract]修饰的方法，这些方法将各自实现委托给父类的Channels属性，从而调用正确的外部方法。下面是这个代理类型的部分代码：

```
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
  "4.5.0.0")]
public partial class EightBallClient :
  System.ServiceModel.ClientBase<IEightBall>, IEightBall
{
...
  public string ObtainAnswerToQuestion(string userQuestion)
  {
    return base.Channel.ObtainAnswerToQuestion(userQuestion);
  }
}
```

创建代理类型的实例时，它的基类将使用客户端应用程序配置文件中的设置来建立与终结点的连接。与服务端配置文件相似，生成的App.config文件包含一个<endpoint>元素和一些关于basicHttpBinding（用来与服务进行通信）的细节。

此外，你会看到如下<client>元素，它再次从客户端角度创建了ABC：

```
<client>
  <endpoint
   address = "http://localhost:8080/MagicEightBallService"
   binding = "basicHttpBinding" bindingConfiguration="BasicHttpBinding_IEightBall"
   contract = "IEightBall" name="BasicHttpBinding_IEightBall" />
</client>
```

至此，你可以把这两个文件放到客户端项目中（并且引用System.ServiceModel.dll程序集），然后使用这个代理类型和远程WCF服务进行通信。但这里我们会采用不同的方式，让我们来看看Visual Studio如何帮助我们自动创建客户端代理文件。

25.9.2　使用Visual Studio生成代理代码

与任何优秀的命令行工具相似，svcutil.exe提供了大量可用于控制如何生成客户端代理的选项。然而，如果不需要使用这些高级选项，你可以使用Visual Studio集成开发环境来生成相同的文件。只需要选择Project菜单的Add Service Reference选项就可以了。

一旦激活了该菜单选项，提示输入服务的URI。单击Go按钮可以查看服务描述，如图25-7所示。

除了向当前项目创建和插入代理文件以外，这个工具还会为我们自动引用WCF程序集。作为命名惯例，代理类在命名空间ServiceReference1中定义，它被嵌套在了客户端命名空间中（为了避免可能的名字冲突）。下面是完整的客户端代码：

```
// 代理所在的位置
using MagicEightBallServiceClient.ServiceReference1;

namespace MagicEightBallServiceClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Ask the Magic 8 Ball *****\n");

      using (EightBallClient ball = new EightBallClient())
      {
        Console.Write("Your question: ");
```

```
        string question = Console.ReadLine();
        string answer =
          ball.ObtainAnswerToQuestion(question);
        Console.WriteLine("8-Ball says: {0}", answer);
      }
      Console.ReadLine();
    }
  }
}
```

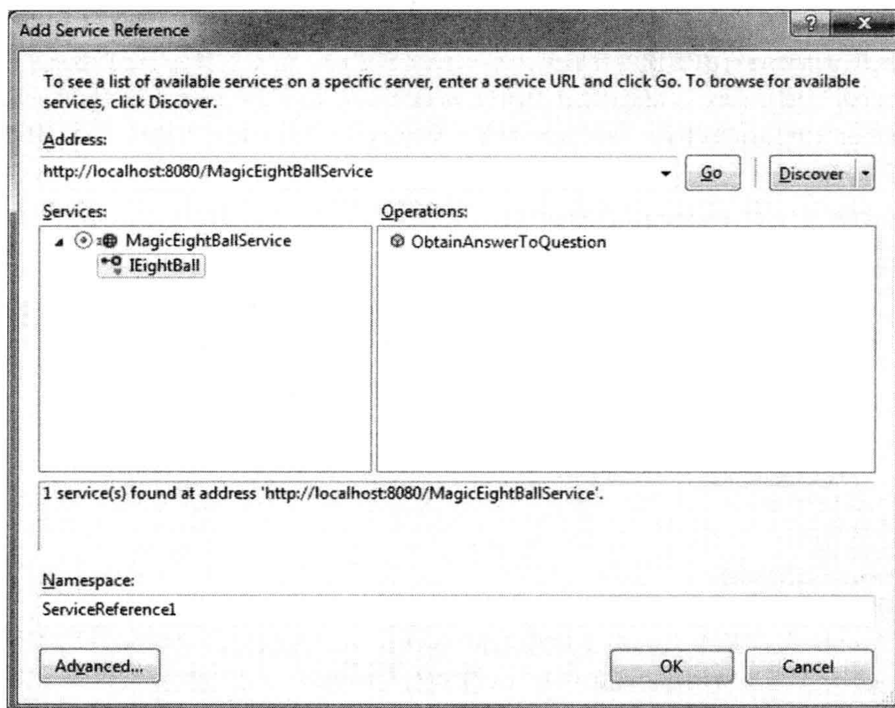


图25-7 使用Visual Studio生成代理代码

现在，假设WCF正在运行，就可以执行客户端。以下是对一个问题可能的响应：

```
***** Ask the Magic 8 Ball *****

Your question: Will I ever finish Skyrim?
8-Ball says: No

Press any key to continue . . .
```

源代码 MagicEightBallServiceClient项目的源代码位于Chapter 25的MagicEightBallServiceHTTP子目录下。

25.9.3　配置基于TCP的绑定

至此，宿主和客户端应用程序都配置为使用最简单的基于HTTP的绑定basicHttpBinding。回忆一下，把设置分离到配置文件的好处是我们可以以声明方式改变基础通道，并且对同一个服务公开多个绑定。

为了演示，让我们做一些实验。在我们的C：驱动器（或者你一直保存代码的地方）上新建一个文件夹EightBallTCP，在这个新文件夹中创建两个子目录Host和Client。

然后，使用Windows资源管理器导航到（本章前面创建的）宿主项目的\bin\Debug文件夹并且将MagicEightBallServiceHost.exe、MagicEightBallServiceHost.exe.config和MagicEightBallServiceLib.dll复制到C:\Eight BallTCP\Host文件夹。使用简单的文本编辑器打开*.config文件进行编辑，然后按如下所示修改既有的内容：

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name= "MagicEightBallServiceLib.MagicEightBallService">
        <endpoint address = ""
                  binding = "netTcpBinding"
                  contract = "MagicEightBallServiceLib.IEightBall"/>
        <host>
          <baseAddresses>
            <add baseAddress = "net.tcp://localhost:8090/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

宿主的*.config文件基本上去除了所有的MEX设置（因为我们已经构建了代理）并且使用netTcpBinding绑定类型进行创建。现在双击*.exe来运行应用程序。如果一切正常的话，你应该能看到如下所示的宿主输出：

```
***** Console Based WCF Host *****

***** Host Info *****
Address: net.tcp://localhost:8090/MagicEightBallService
Binding: NetTcpBinding
Contract: IEightBall

**********************
The service is ready.
Press the Enter key to terminate service.
```

为了完成这个测试，把MagicEightBallServiceClient.exe和MagicEightBallServiceClient.exe.config文件从（本章前面创建的）客户端应用程序的\bin\Debug文件夹复制到C:\EightBallTCP\Client文件夹。按如下所示更新客户端配置文件：

```
<?xml version= "1.0" encoding= "utf-8" ?>
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address = "net.tcp://localhost:8090/MagicEightBallService"
                binding = "netTcpBinding"
                contract = "ServiceReference1.IEightBall"
                name = "netTcpBinding_IEightBall" />
    </client>
  </system.serviceModel>
</configuration>
```

这个客户端配置文件是Visual Studio代理生成器生成的极度简化形式。注意，我们完全移除了既有的<bindings>元素。起初，*.config文件包含了<bindings>元素和<basicHttpBinding>子元素，它们提供了客户端绑定设置的许多细节（超时等）。

实际上，我们的示例不需要这些细节，因为我们自动获取基础BasicHttpBinding对象的默认值。如果需要，我们当然可以更新既有的<bindings>元素来定义<netTcpBing>子元素的细节。但是如果NetTcpBinding对象默认值可以满足要求，就不需要这样做。

不管怎么样，你现在应该可以运行客户端应用程序。如果宿主还在后台运行的话，就可以使用TCP在程序集之间转移数据。

源代码 MagicEightBallTCP项目的源代码位于Chapter 25子目录下。

25.10 简化配置设置

在学习本章的第一个示例时，你会发现宿主的配置逻辑相当烦琐。例如，宿主的*.config文件（基本HTTP绑定）需要为服务和MEX分别定义一个<endpoint>元素，为减少冗余URI定义一个<baseAddresses>元素（这是可选的），为定义元数据交换的运行时性质要配置<behaviors>节。

显然，学习如何编写宿主的*.config文件成为构建WCF服务的主要障碍。更糟的是，很多WCF服务都要求在宿主配置文件中包含相同的基本配置。例如，创建一个全新的WCF服务机器宿主，使用<basicHttpBinding>和MEX公开该服务，所需的*.config文件看上去与前面编写的几乎完全相同。

幸好从.NET 4.0开始，WCF API发布了一些简化方法，包括简化构建宿主配置文件过程的默认配置（及其他快捷方式）。

25.10.1 使用默认终结点

在不支持默认终结点之前，如果没有在配置文件中指定<endpoint>元素，就调用ServieHost对象的Open()的话，将抛出运行时异常。如果在代码中调用AddServiceEndpoint()来指定终结点，也将得到相同的结果。但在.NET 4.5中，每个WCF服务都自动提供了*默认终结点*，为每个支持的协议捕获普通的配置细节。

打开.NET 4.5中的machine.config文件，将发现一个新的元素<protocolMapping>。该元素列出了未指定WCF绑定时将使用的默认绑定：

```
<system.serviceModel>
...
  <protocolMapping>
    <add scheme = "http" binding= "basicHttpBinding"/>
    <add scheme = "net.tcp" binding= "netTcpBinding"/>
    <add scheme = "net.pipe" binding= "netNamedPipeBinding"/>
    <add scheme = "net.msmq" binding= "netMsmqBinding"/>
  </protocolMapping>
    ...
</system.serviceModel>
```

要使用这些默认绑定，你要做的仅仅是在宿主配置文件中指定基地址。在Visual Studio中打开基于HTTP的MagicEightBallServiceHost项目。要更新*.config文件，可以删除所有WCF服务的`<endpoint>`元素和MEX指定的数据。现在配置文件如下所示：

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

由于指定了一个有效的HTTP `<baseAddress>`，宿主程序将自动使用`basicHttpBinding`。再次运行宿主程序，将得到相同的ABC数据：

```
***** Console Based WCF Host *****

***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall
**********************
The service is ready.
Press the Enter key to terminate service.
```

我们还没有启用MEX，稍后将使用另一个简化方法——默认行为配置来启用MEX。不过，我们首先要学习的是如何使用多重绑定公开单独的WCF服务。

25.10.2　使用多重绑定公开单独的WCF服务

从最初发布开始，WCF就支持单独的宿主通过多重终结点公开WCF服务。例如，通过在配置文件中添加新的终结点，你可以使用HTTP、TCP和命名管道绑定来公开`MagicEightBallService`。重启宿主程序，将自动创建所有必要的通道。

在WCF之前，使用多个绑定公开单独的服务是很困难的，因为不同类型的绑定（HTTP或TCP）包含不同的编程模型。然而，允许调用者选择最适合的绑定是很有用的。内部调用者可能希望使用TCP

绑定，外部客户端（在公司防火墙之外）可能需要使用HTTP来进行访问，而同一台机器上的客户端则会使用命名管道。

在.NET 4.5之前，需要手工在宿主配置文件中定义多个<endpoint>元素。还可能要为每个协议定义多个<baseAddress>元素。但是，现在只需要编写下面的配置文件：

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
            <add baseAddress =
                "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

编译项目（这样可以刷新部署的*.config文件）并重启宿主程序，可以看到如下的终结点数据：

```
***** Console Based WCF Host *****

***** Host Info *****
Address: http://localhost:8080/MagicEightBallService
Binding: BasicHttpBinding
Contract: IEightBall

Address: net.tcp://localhost:8099/MagicEightBallService
Binding: NetTcpBinding
Contract: IEightBall

**********************
The service is ready.
Press the Enter key to terminate service.
```


现在WCF服务可从两种不同的终结点到达，那么调用者如何在两者之间进行选择呢？在生成客户端代理时，Add Service Reference工具将在客户端*.config文件中为每个公开的终结点起一个字符串形式的名称。在代码中，可以向代理的构造函数传递正确的字符串名称，并且肯定，会使用正确的绑定。在此之前，我们需要为修改后的宿主配置文件重新建立MEX，并学习如何调整默认绑定的设置。

25.10.3 修改WCF绑定的设置

如果在C#代码中指定了服务的ABC（本章稍后将介绍），那么如何修改WCF绑定的默认设置就显而易见了，只需要修改对象的属性值即可。例如，如果要使用BasicHttpBinding并希望更改超时设置，可以使用如下代码：

```
void ConfigureBindingInCode()
{
  BasicHttpBinding binding = new BasicHttpBinding();
```

```
  binding.OpenTimeout = TimeSpan.FromSeconds(30);
  ...
}
```

你还可以以声明方式配置绑定的设置。例如，.NET 3.5可以构建修改了BasicHttpBinding的OpenTimeout属性的宿主配置文件，如下所示：

```
<configuration>
  <system.serviceModel>

    <bindings>
      <basicHttpBinding>
        <binding name = "myCustomHttpBinding"
                 openTimeout = "00:00:30" />
      </basicHttpBinding>
    </bindings>

    <services>
      <service name = "WcfMathService.MyCalc">
        <endpoint address = "http://localhost:8080/MyCalc"
                  binding = "basicHttpBinding"
                 bindingConfiguration = "myCustomHttpBinding"
                  contract = "WcfMathService.IBasicMath" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

这里，我们为支持IBasicMath接口的WcfMathService.MyCalc服务编写了一个配置文件。注意，在<bindings>节中可以定义命名的<binding>元素，该元素可用来调整给定绑定的设置。在服务的<endpoint>内，可以使用bindingConfiguration特性来连接指定的设置。

这种宿主配置方式同样有效。但若使用默认的终结点，就不能将<binding>连接到<endpoint>了。幸好你可以省略<binding>元素的name特性，来控制默认终结点的设置。例如，下面的标记片段修改了后台使用的默认BasicHttpBinding和NetTcpBinding对象的某些属性：

```
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MagicEightBallServiceLib.MagicEightBallService" >
        <host>
          <baseAddresses>
            <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
            <add baseAddress =
              "net.tcp://localhost:8099/MagicEightBallService"/>
          </baseAddresses>
        </host>
      </service>
    </services>

    <bindings>
      <basicHttpBinding>
        <binding openTimeout = "00:00:30" />
      </basicHttpBinding>
      <netTcpBinding>
        <binding closeTimeout ="00:00:15"/>
      </netTcpBinding>
    </bindings>
```

```
  </system.serviceModel>
</configuration>
```

25.10.4 使用默认的MEX行为配置

代理生成工具必须在运行时发现服务，才能开始工作。在WCF中，可以通过启用MEX来允许这种运行时发现。而且，大多数宿主配置文件都需要启用MEX（至少在开发时是这样）。幸好我们很少需要修改MEX配置，因此.NET 4.5提供了一些快捷方式。

最常用的快捷方式是默认的MEX支持。你不需要添加MEX终结点、定义命名的MEX服务行为并将命名的绑定连接到服务（像HTTP版本的MagicEightBallServiceHost那样），而只需要添加如下标记：

```
<configkration>
<system.serviceModel>
  <services>
    <service name = "MagicEightBallServiceLib.MagicEightBallService" >
      <host>
        <baseAddresses>
          <add baseAddress = "http://localhost:8080/MagicEightBallService"/>
          <add baseAddress =
            "net.tcp://localhost:8099/MagicEightBallService"/>
        </baseAddresses>
      </host>
    </service>
  </services>

  <bindings>
    <basicHttpBinding>
      <binding openTimeout = "00:00:30" />
    </basicHttpBinding>
    <netTcpBinding>
      <binding closeTimeout ="00:00:15"/>
    </netTcpBinding>
  </bindings>

  <behaviors>
    <serviceBehaviors>
      <behavior>
        <!--  不要命名<serviceMetadata>元素，这样可以得到默认的MEX -->
        <serviceMetadata httpGetEnabled = "true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>

  </system.serviceModel>
</configuration>
```

关键点是<serviceMetadata>元素不再设置name特性（并且<service>元素也不再设置behaviorConfiguration特性）。这样就可以在运行时得到默认的MEX支持。编译并刷新配置文件，然后运行宿主程序，在浏览器中输入下面的URL：

```
http://localhost:8080/MagicEightBallService
```

打开之后，单击页面上方的wsdl链接，查看服务的WSDL描述（如图25-6所示）。注意，宿主程序的控制台窗口不会打印MEX终结点的数据，因为我们没有在配置文件中为IMetadataExchange显式地定

义终结点。但MEX已经启用了，可以开始构建客户端代理了。

25.10.5 刷新客户端代理和选择绑定

假设更新后的宿主程序已经编译并在后台运行，你现在可能希望打开客户端应用程序并刷新当前的服务引用。在Solution Explorer中打开Service References文件夹，右击当前的ServiceReference1，选择Update Service Reference菜单选项（如图25-8所示）。

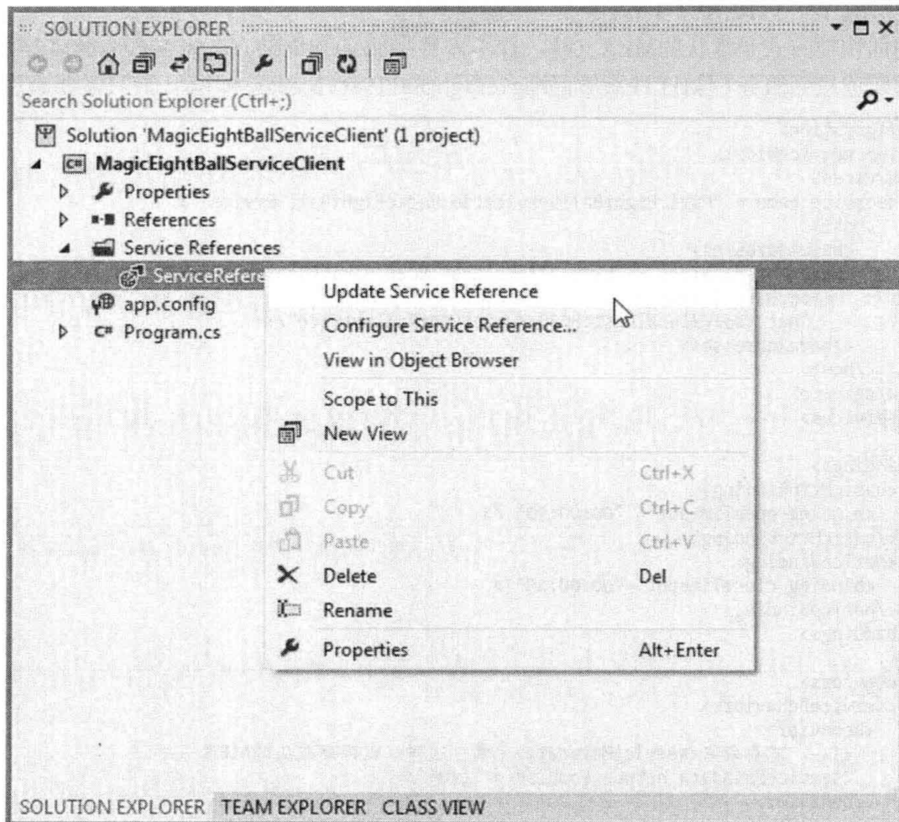


图25-8　刷新代理和客户端的*.config文件

完成之后，将在客户端的*.config文件中看到两个可选的绑定：一个用于HTTP，一个用于TCP。可以看到，每个绑定都有一个适当的名称。以下是刷新后配置文件的部分内容：

```
<configuration>
  <system.serviceModel>

    <bindings>
      <basicHttpBinding>
        <binding name = "BasicHttpBinding_IEightBall" ... />
      </basicHttpBinding>
```

```
      <netTcpBinding>
        <binding name = "NetTcpBinding_IEightBall" ... />
      </netTcpBinding>
    </bindings>
...
    </system.serviceModel>
</configuration>
```

客户端在创建代理对象并选择要使用的绑定时，可以使用这些名称。因此，如果客户端要使用TCP，可以使用如下的C#代码：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Ask the Magic 8 Ball *****\n");

  using (EightBallClient ball = new EightBallClient("NetTcpBinding_IEightBall"))
  {
    ...
  }
  Console.ReadLine();
}
```

如果客户端要使用HTTP绑定，可以使用如下的代码：

```
using (EightBallClient ball = new
       EightBallClient("BasicHttpBinding_IEightBall"))
{
...
}
```

在当前的示例中，我们介绍了很多有用的快捷方式。这些特性简化了编写宿主配置文件的方式。接下来，你将看到如何使用WCF Service Library Project模板。

源代码　MagicEightBallServiceHTTPDefaultBindings项目的源代码位于Chapter 25子目录下。

25.11　使用 WCF 服务库项目模板

在我们构建WCF服务来和第21章中创建的AutoLot数据库进行通信之前，下一个示例会阐明许多重要的主题，包括WCF服务库项目模板的优势、WCF测试客户端、WCF配置编辑器、把WCF服务承载在Windows服务中以及异步客户端调用。为了便于关注这些新的方面，WCF服务仍然会很简单。

25.11.1　构建简单的Math服务

首先，创建一个叫MathServiceLibrary的全新WCF服务库项目，请确保在New Project对话框的WCF节点下选择正确的选项（如果需要提示的话，请看图25-2）。现在将初始IService1.cs文件名改为IBasicMath.cs。完成后，删除MathServiceLibrary命名空间中所有的示例代码，然后使用如下代码进行替换：

```
[ServiceContract(Namespace="http://MyCompany.com")]
public interface IBasicMath
{
```

```
  [OperationContract]
  int Add(int x, int y);
}
```

然后，将Service1.cs文件名改为MathService.cs，并且再一次删除MathServiceLibrary命名空间中的所有示例代码，然后这样实现服务契约：

```
public class MathService : IBasicMath
{
  public int Add(int x, int y)
  {
    // 为了模拟长请求
    System.Threading.Thread.Sleep(5000);
    return x + y;
  }
}
```

最后，打开提供的App.config文件并将所有的IService1修改为IBasicMath，将所有的Service1修改为MathService。同样，注意*.config文件已经启用了对MEX的支持，并且默认使用的是wsHttpBinding协议。

25.11.2　使用WcfTestClient.exe测试WCF服务

使用WCF服务库项目的优势体现在调试或运行库的时候，它会从*.config文件中读取设置，然后用它们加载WCF测试客户端应用程序（WcfTestClient.exe）。这个基于GUI的应用程序允许我们在构建WCF服务的时候测试服务接口的每个成员，而无须像之前那样为了测试而手动构建宿主/客户端。

图25-9显示了MathService的测试环境。注意，双击一个接口方法，就可以指定输入参数并且调用成员。

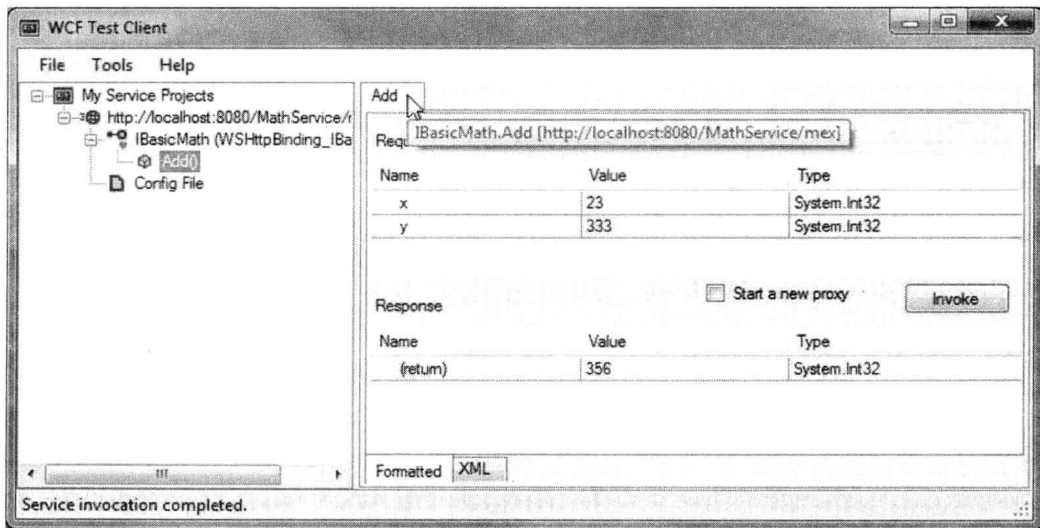


图25-9　使用WcfTestClient.exe测试WCF服务

虽然在创建WCF服务库项目的时候，这个工具是现成的，但是要知道如果通过指定MEX终结点在命令行中启动它的话，我们可以用它来测试任何WCF服务。例如，如果要启动MagicEightBall ServiceHost.exe应用程序，就可以在Developer Command Prompt中指定如下命令：

```
wcftestclient http://localhost:8080/MagicEightBallService
```

完成后，我们就可以以相似方式调用ObtainAnswerToQuestion()。

25.11.3　使用SvcConfigEditor.exe修改配置文件

使用WCF服务库项目的另外一个好处是，我们可以在Solution Explorer中右击App.config文件来激活基于GUI的服务配置编辑器SvcConfigEditor.exe（如图25-10所示）。引用WCF服务的客户端应用程序也可以使用相同的技术。

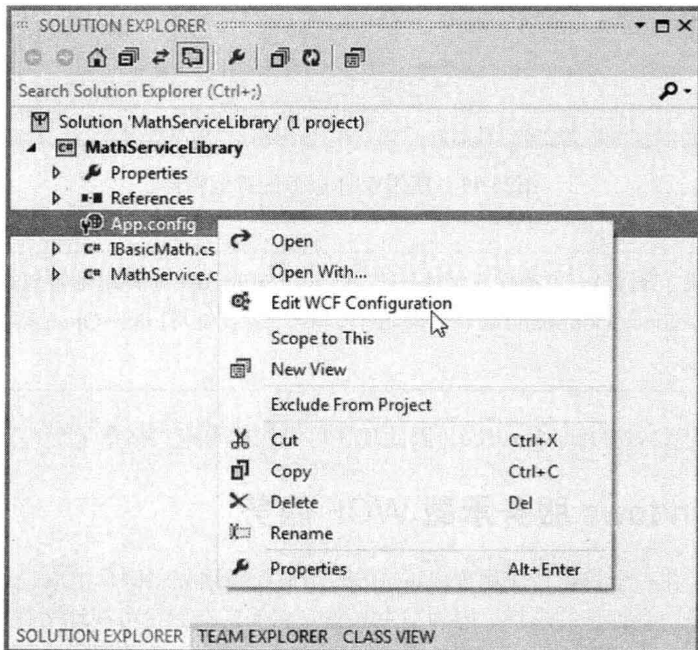


图25-10　从这里开始基于GUI的*.config文件编辑

激活这个工具之后，我们就可以使用友好的用户界面来改变基于XML的数据。使用像这样的工具来维护我们的*.config文件有很多明显的好处。首先，我们可以确保生成的标记遵守期望的格式并且不会有输入错误。其次，还可以看到可以赋给某个属性的有效值。最后，我们不需要手动编写乏味的XML数据。

图25-11演示了服务配置编辑器的总体外观。说实话，可以用整章的内容来描述SvcConfigEditor. exe支持的有趣选项（COM+集成、新建*.config文件等）。请一定要花时间研究这个工具，要知道你可以通过按F1键来获得相对详细的帮助系统。

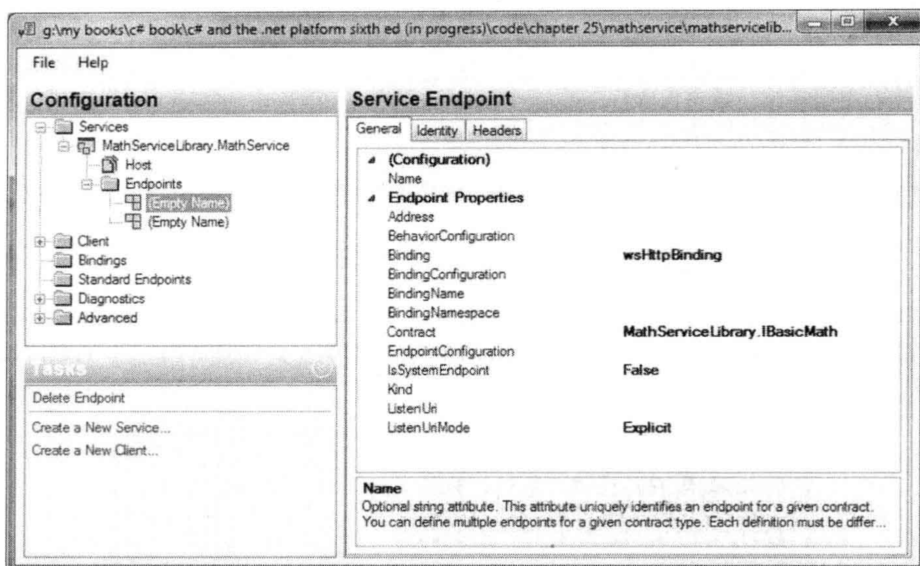


图25-11 使用WCF服务配置编辑器

说明 即使你开始没有选择WCF服务库项目，SvcConfigEditor.exe工具还是可以编辑（或创建）配置文件。使Developer Command窗口启动这个工具，然后使用File→Open菜单项来加载要编辑的既有*.config。

不需要进一步配置WCF MathService，因此现在可以构建自定义宿主了。

25.12 以 Windows 服务承载 WCF 服务

你可能会同意，在一个控制台应用程序（或在GUI桌面应用程序中）中承载WCF服务对于产品级别的服务器来说不是一个理想的选择，因为宿主必须一直保持在后台可视地运行来服务客户端。即使我们把宿主应用程序最小化到Windows任务栏中，仍然有可能不小心关闭这个程序，这样也就终止了任何客户端应用程序的连接。

说明 虽然桌面Windows应用程序不一定要显示主窗口，但是大多数*.exe需要用户的交互才能加载可执行程序。因此，Windows服务（后面会介绍）可以配置为即使用户当前没有登录到工作站的时候也能运行。

如果你正在构建一个内部的WCF应用程序，另外一种承载WCF服务库的方案就是使用专有的Windows服务。这样做的一个好处是Windows服务可以配置为在目标机器启动的时候自动启动。另一个好处是Windows服务在后台不可视地运行（和控制台应用程序不同）并且不需要用户交互性（而且

不需要在宿主计算机上安装IIS)。

为了演示如何构建这样的宿主,首先新建一个叫MathWindowsServiceHost的Windows服务项目(如图25-12所示)。完成后,使用Solution Explorer将最初的Service1.cs文件重命名为MathWinService.cs。

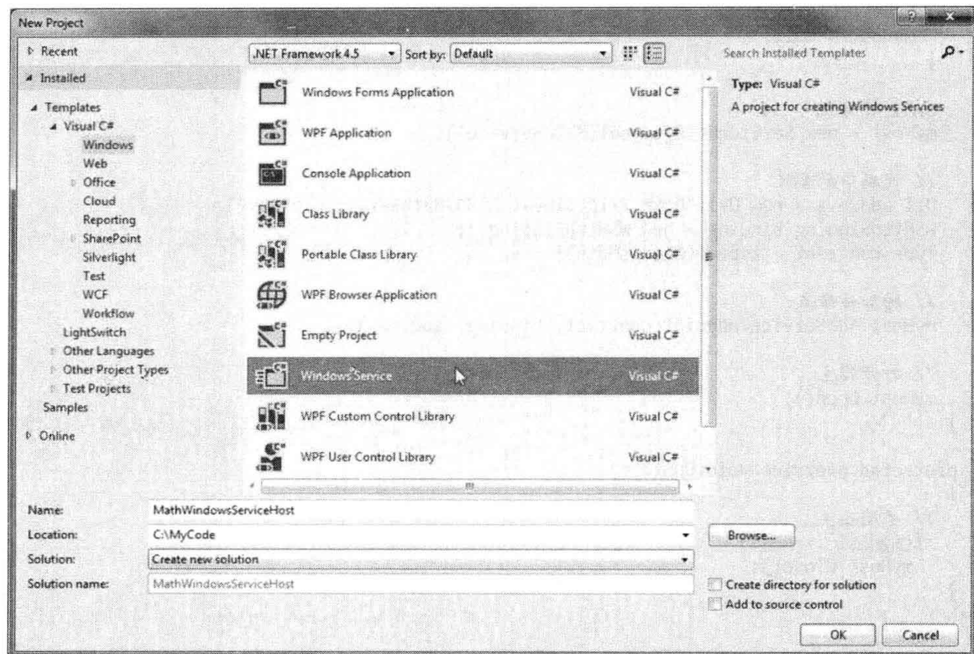


图25-12 创建一个Windows服务来承载我们的WCF服务

25.12.1 在代码中指定ABC

现在,假设我们设置了MathServiceLibrary.dll和System.ServiceModel.dll程序集的引用,所有需要做的就是在Windows服务类型的OnStart()和OnStop()中使用ServiceHost类型。打开服务宿主类的代码文件(通过右击设计器并选择View Code)并且增加如下逻辑:

```
// 确保导入这些命名空间
using MathServiceLibrary;
using System.ServiceModel;

namespace MathWindowsServiceHost
{
  public partial class MathWinService: ServiceBase
  {
    // ServiceHost类型的成员变量
    private ServiceHost myHost;

    public MathWinService()
    {
      InitializeComponent();
    }
```

```
    protected override void OnStart(string[] args)
    {
      // 只是为了确保安全
      if (myHost != null)
      {
        myHost.Close();
        myHost = null;
      }

      // 创建宿主
      myHost = new ServiceHost(typeof(MathService));

      // 代码中的ABC
      Uri address = new Uri("http://localhost:8080/MathServiceLibrary");
      WSHttpBinding binding = new WSHttpBinding();
      Type contract = typeof(IBasicMath);

      // 增加终结点
      myHost.AddServiceEndpoint(contract, binding, address);

      // 打开宿主
      myHost.Open();
    }

    protected override void OnStop()
    {
      // 关闭宿主
      if(myHost != null)
        myHost.Close();
    }
  }
}
```

虽然在为WCF服务构建Windows服务宿主的时候，完全可以使用配置文件，但是在这里（为了改变习惯）我们以编程方式使用Uri、WSHttpBinding和Type类来创建终结点，而不是使用*.config文件。在创建了ABC的每个方面之后，我们就会通过调用AddServiceEndpoint()来以编程方式通知宿主。

如果想通知运行时，你希望访问.NET 4.5 machine.config 文件中的所有默认终结点绑定，可以简化编程逻辑，在调用ServiceHost构造函数的时候指定基地址。在这种情况下，就不需要在代码中手工指定ABC或调用AddServiceEndpoint()，而只需要调用AddDefaultEndpoints()。考虑如下的更新：

```
protected override void OnStart(string[] args)
{
  if (myHost != null)
  {
    myHost.Close();
  }
  // 创建宿主，并为HTTP绑定指定URL
  myHost = new ServiceHost(typeof(MathService),
           new Uri("http://localhost:8080/MathServiceLibrary"));

  // 选择默认的终结点
  myHost.AddDefaultEndpoints();

  // 打开宿主
  myHost.Open();
}
```

25.12.2 启用MEX

虽然我们也可以以编程方式开启MEX，但是我们会采用配置文件。向Windows服务项目插入一个新的App.config文件，它包含如下的MEX设置：

```
<?xml version = "1.0" encoding = "utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name = "MathServiceLibrary.MathService">
    </service>
  </services>

  <behaviors>
    <serviceBehaviors>
      <behavior>
        <serviceMetadata httpGetEnabled = "true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>

  </system.serviceModel>
</configuration>
```


25.12.3 创建Windows服务安装程序

为了把Windows服务注册到操作系统，我们需要向项目增加一个安装程序来包含注册服务的必要代码。只需要右击Windows服务设计器界面并且选择Add Installer（如图25-13所示）即可。

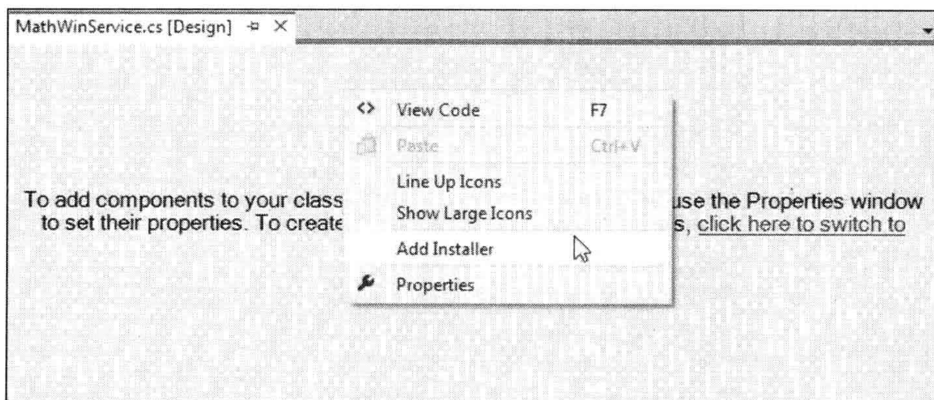


图25-13　为Windows服务增加安装程序

完成后，你就会看到新设计器界面上加入了两个组件。第一个组件（默认为serviceProcessInstaller1）表示可以在目标机器上安装的新的Windows服务的类型。在设计器上选择这个类型，然后在Properties窗口中将Account属性设置为LocalSystem（如图25-14所示）。

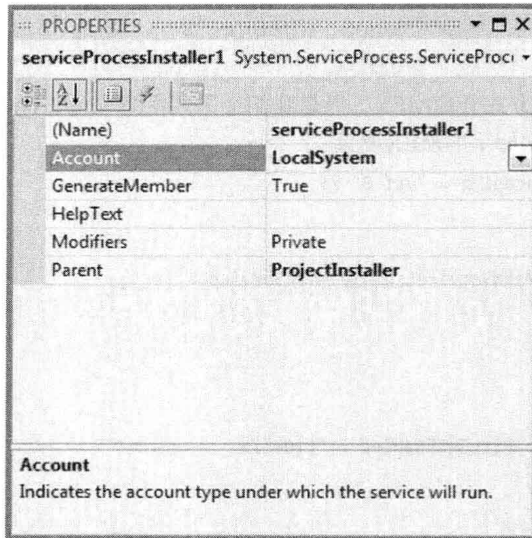


图25-14　确保以本地系统账号运行Windows服务

第二个组件（名为`serviceInstaller1`）表示会安装我们的某个Windows服务的类型。同样，使用Properties窗口将`ServiceName`属性修改为`MathService`，将`StartType`属性设置为`Automatic`，并且通过`Description`属性增加一个Windows服务的友好描述（如图25-15所示）。

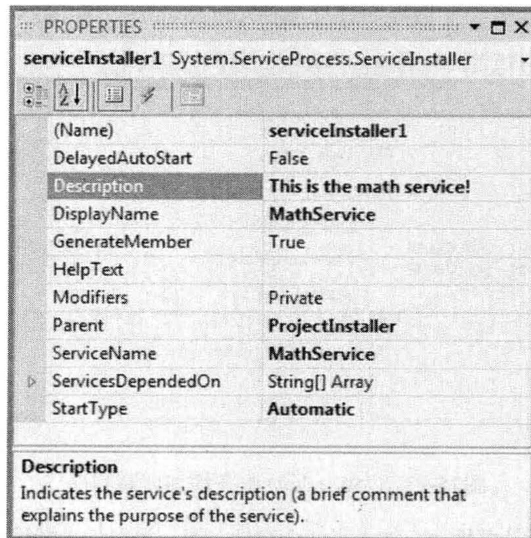


图25-15　配置安装器细节

至此，我们就可以编译应用程序了。

25.12.4 安装Windows服务

Windows服务可以使用传统的安装程序（如*.msi安装程序）或通过installutil.exe命令行工具来安装到宿主上。

说明 要使用intallutil.exe安装Windows服务，必须以管理员权限启动Developer Command Prompt。右击Developer Command Prompt图标，选择Run As Administrator。

使用Developer Command Prompt，转到MathWindowsServiceHost项目的\bin\Debug文件夹。现在输入如下的命令：

```
installutil MathWindowsServiceHost.exe
```

如果安装成功，现在就可以打开控制面板中Administrative Tools文件夹下的服务工具。可以看到Windows服务的友好名称已经按照字母顺序列出了。找到它后，就可以使用Start链接启动我们本地机器上的服务（如图25-16所示）。

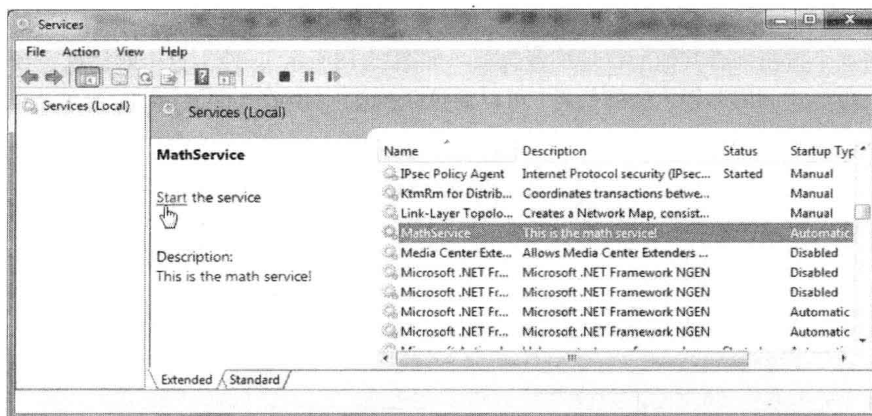


图25-16 查看我们的Windows服务，它承载了我们的WCF服务

既然服务存在并且启动了，最后一步就是构建客户端应用程序来消费这个服务。

源代码 MathWindowsServiceHost项目的源代码位于Chapter 25子目录下。

25.13 从客户端异步调用服务

新建一个控制台应用程序项目MathClient，然后使用Visual Studio的Add Service Reference选项来添加运行中WCF服务（当前由后台运行的Windows服务承载）的引用（需要在Addresses框中键入URL，可能为http://localhost:8080/MathServiceLibrary）。然而，暂时不要单击OK按钮。注意Add Service Reference对话框中左下角的Advanced按钮（如图25-17所示）。

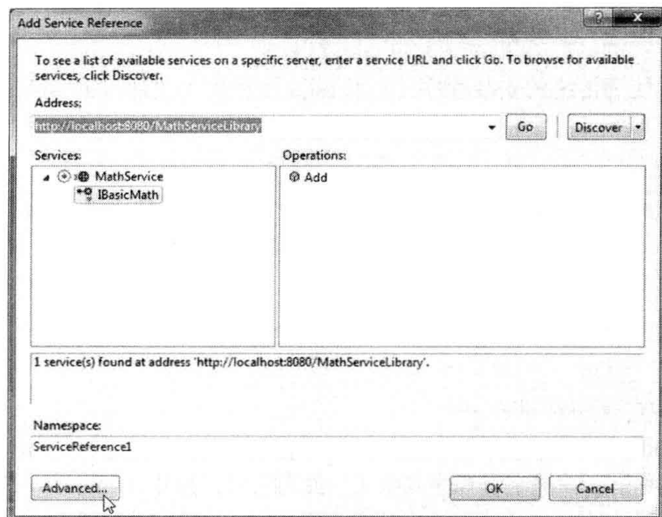


图25-17　引用MathService并准备配置高级设置

现在单击这个按钮来看其他的代理配置选项（如图25-18所示）。使用这个对话框，我们可以生成允许我们以异步形式调用远程方法的代码，可以通过Generate asynchronous operators单选框来完成。现在，请选中这个选项。

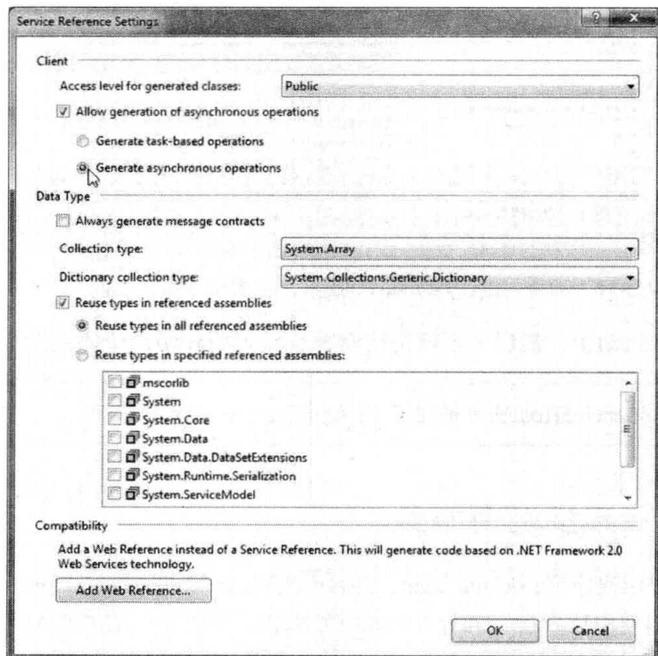


图25-18　高级客户端代理配置选项

至此，代理代码就会包含允许我们使用第19章介绍的Begin/End异步调用方式调用每个服务契约成员。这里使用的是一个使用Lambda表达式而不是强类型AsyncCallback委托的简单实现。

```
using System;
using MathClient.ServiceReference1;
...

namespace MathClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** The Async Math Client *****\n");

      using (BasicMathClient proxy = new BasicMathClient())
      {
        proxy.Open();
        // 使用Lambda表达式以异步方式增加数字
        IAsyncResult result = proxy.BeginAdd(2, 3,
          ar =>
          {
            Console.WriteLine("2 + 3 = {0}", proxy.EndAdd(ar));
          },
          null);

        while (!result.IsCompleted)
        {
          Thread.Sleep(200);
          Console.WriteLine("Client working...");
        }
      }
      Console.ReadLine();
    }
  }
}
```

源代码 MathClient项目的源代码位于Chapter 25子目录下。

25.14 定义 WCF 数据契约

本章最后一个示例是WCF数据契约的构建。之前的WCF服务定义了非常简单的方法操作原生CLR数据类型。只要使用任何HTTP绑定类型（basicHttpBinding、wsHttpBinding等），传入的和输出的数据类型都会被自动格式化为XML元素。补充一点，如果使用的是基于TCP的绑定（如netTcpBinding），简单数据类型的参数和返回值就会用紧凑二进制格式进行传输。

说明 WCF运行库还会自动为所有使用[Serializable]的特性进行编码。但这并不是定义WCF契约的推荐方法，它只是为了向后兼容。

然而，如果我们定义服务契约将自定义类型作为参数或返回值，那么最佳实践是用WCF数据契约对这些数据进行建模。简而言之，数据契约是使用[DataContract]特性的类型。每一个需要用作契约的一部分的字段也标记了[DataMember]特性。

说明　在早期的.NET平台版本中，要确保正确表示自定义数据类型，就要强制使用[DataContract]和[DataMember]。从技术上说，在自定义数据类型上使用这些特性不是必需的。但这被认为是一项.NET最佳实践。

25.14.1　使用Web相关的WCF服务项目模板

我们的下一个WCF服务将允许外部调用者和第21章中创建的AutoLot数据库进行交互。此外，这个最后的WCF服务会使用基于Web的WCF服务模板来创建并部署在IIS下。

打开Visutal Studio（以管理员权限），访问File→New→Web Site菜单项。选择WCF Service项目类型，并确保Web Location的下拉框选择的是HTTP（这将在IIS中安装我们的服务）。将服务公开为下面的URI：

http://localhost/AutoLotWCFService

图25-19展示了配置后的项目。

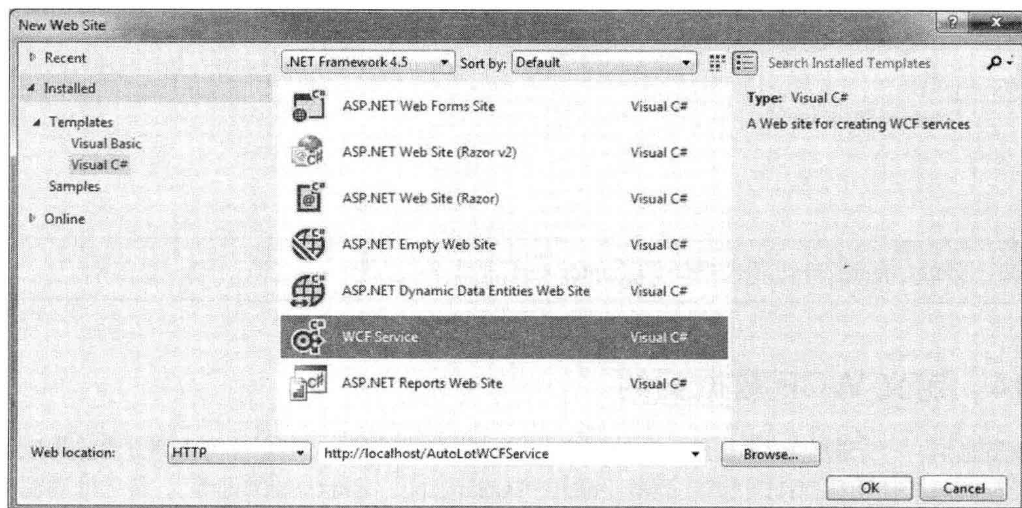


图25-19　创建Web相关的WCF服务

完成后，添加第21章中创建的AutoLotDAL.dll程序集的引用（通过Website→Add Reference菜单项）。有一些示例初始代码（在App_Code文件夹中），你可能会要删除。首先，将最初的IService.cs文件重命名为IAutoLotService.cs，然后在新命名的文件中定义最初的服务契约，如下所示：

```
[ServiceContract]
public interface IAutoLotService
```

```
{
  [OperationContract]
  void InsertCar(int id, string make, string color, string petname);

  [OperationContract]
  void InsertCar(InventoryRecord car);

  [OperationContract]
  InventoryRecord[] GetInventory();
}
```

这个接口定义了3个方法，其中一个返回（还未创建的）InventoryRecord类型的数组。回忆一下，InventoryDAL的GetInventory()方法只是返回DataTable对象，你可能会问为什么我们服务的GetInventory()方法不这么做呢。

虽然从WCF服务方法返回DataTable是可行的，但是回忆一下WCF的构建遵循SOA标准，其中一个要点就是对契约编程，而不是实现。因此，我们会返回以未知方式用WSDL文档正确表达的自定义数据契约（InventoryRecord），而不会把.NET特有的DataTable类型返回给外部调用者。

还要注意，接口定义了叫做InsertCar()的重载方法。第一个版本接受了4个传入参数，而第二个版本接受了InventoryRecord类型作为输入。你可以按照如下所示定义InventoryRecord数据契约：

```
[DataContract]
public class InventoryRecord
{
  [DataMember]
  public int ID;

  [DataMember]
  public string Make;

  [DataMember]
  public string Color;

  [DataMember]
  public string PetName;
}
```

如果像现在这样实现IAutoLotService接口，编译宿主并且尝试从客户端调用这些方法的话，你可能会得到一个运行时异常。出现这个问题的原因是WSDL描述要求每一个从给定终结点公开的方法必须具有唯一的名字。因此，虽然对于C#来说方法重载可以实现，但是当前的Web服务规范不允许两个方法都命名为InsertCar()。

可喜的是，[OperationContract]特性支持一个命名属性（Name），它能让我们指定如何在WSDL描述中表示C#方法。那么，可以按如下所示更新第二个版本的InsertCar()：

```
public interface IAutoLotService
{
...
  [OperationContract(Name = "InsertCarWithDetails")]
  void InsertCar(InventoryRecord car);
}
```

25.14.2 实现服务契约

现在将Service.cs重命名为AutoLotService.cs。AutoLotService类型按如下方式实现了IAutoLot-

Service接口（请确保在代码文件中导入AutoLotConnectedLayer和System.Data命名空间，并在需要的时候更新连接字符串）:

```
using AutoLotConnectedLayer;
using System.Data;

public class AutoLotService : IAutoLotService
{
  private const string ConnString =
    @"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot"+
     ";Integrated Security=True";

  public void InsertCar(int id, string make, string color, string petname)
  {
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    d.InsertAuto(id, color, make, petname);
    d.CloseConnection();
  }

  public void InsertCar(InventoryRecord car)
  {
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    d.InsertAuto(car.ID, car.Color, car.Make, car.PetName);
    d.CloseConnection();
  }

  public InventoryRecord[] GetInventory()
  {
    // 首先，从数据库获取DataTable
    InventoryDAL d = new InventoryDAL();
    d.OpenConnection(ConnString);
    DataTable dt = d.GetAllInventoryAsDataTable();
    d.CloseConnection();

    // 创建List<T>来获取记录
    List<InventoryRecord> records = new List<InventoryRecord>();

    // 把数据表复制到客户端契约的List<>中
    DataTableReader reader = dt.CreateDataReader();
    while (reader.Read())
    {
      InventoryRecord r = new InventoryRecord();
      r.ID = (int)reader["CarID"];
      r.Color = ((string)reader["Color"]);
      r.Make = ((string)reader["Make"]);
      r.PetName = ((string)reader["PetName"]);
      records.Add(r);
    }

    // 把List<T>转换到InventoryRecord类型的数组
    return (InventoryRecord[])records.ToArray();
  }
}
```

没什么可说的。为了简单，我们硬编码了连接字符串值（可能需要根据机器配置进行调整），而不是把它保存在web.config文件中。由于我们的数据访问类库确实和AutoLost数据库进行真实的通信，

我们需要做的只是把参数传入InventoryDAL类类型的InsertAuto()方法中。还有一个有趣的地方是，把DataTable对象的值映射到泛型InventoryRecord类型（使用DataTableReader来实现）列表，然后把List<T>转换到InventoryRecord类型的数组。

25.14.3 *.svc文件的作用

在创建Web相关的WCF服务时，你会发现项目包含了一个具有*.svc文件扩展名的特殊文件。这个特殊文件对于任何由IIS承载的WCF服务来说都是必需的，它描述了安装点服务实现的名字和位置。因为我们改变了起始文件和WCF类型的名字，所以必须按如下所示更新Service.svc文件的内容：

```
<%@ ServiceHost Language="C#" Debug="true"
      Service="AutoLotService" CodeBehind="~/App_Code/AutoLotService.cs" %>
```

25.14.4 更新web.config文件

在HTTP下创建的WCF服务的web.config文件将使用很多本章前面介绍的WCF简化配置。本书稍后研究ASP.NET的时候会更详细介绍，web.config文件和可执行的*.config文件有相似的作用。但它还控制了许多Web特有的设置。如下面的示例启用了MEX，并且不需要手工指定自定义的<endpoint>：

```
<configuration>

  ...

  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <!-- 为了避免泄露元数据信息，在部署前要将下面的值设置为false，并移除元数据终结点 -->
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true"/>
          <!-- 为了在调试时接收异常信息，可以将下面的值设置为true。在部署前将其设置为false，
              可以避免泄露异常信息-->
          <serviceDebug includeExceptionDetailInFaults="false"/>
        </behavior>
      </serviceBehaviors>
</behaviors>
<serviceHostingEnvironment aspNetCompatibilityEnabled="true"
                           multipleSiteBindingsEnabled="true" />
  </system.serviceModel>

...

</configuration>
```

25.14.5 测试服务

现在我们完全可以构建任何形式的客户端来测试服务，当然也可以把*.svc文件的终结点传入WcfTestClient.exe应用程序：

```
WcfTestClient http://localhost/AutoLotWCFService/Service.svc
```

如果要构建一个自定义客户端应用程序，可以通过Add Service Reference对话框来完成，所用的方法和本章前面的MagicEightBallServiceClient项目和MathClient项目一样。

源代码　AutoLotService项目的源代码位于Chapter 25子目录下。

对WCF API的学习至此告一段落。当然，关于WCF还有很多内容没有涉及，但如果你了解了本章所讲述的内容，就可以为更深入的学习打下良好的基础。要学习WCF的更多内容，请参考.NET Framework 4.5 SDK文档。

25.15　小结

本章介绍了.NET 3.0及其更高版本所提供的WCF API。正如前面所说的，WCF背后的主要动机是提供一个统一的对象模型。该模型使用单一的编程接口来提供一些（在以前是不相关的）分布式计算API。此外，WCF服务是使用特定的地址、绑定和契约来表示的（可以通过使用友好的缩略语ABC来方便记忆）。

一个典型的WCF应用程序包括对三个相关程序集的使用。第一个程序集定义了服务契约和代表服务功能的服务类型。然后，该程序集可由自定义可执行文件、IIS虚拟目录、Windows服务所承载。最后，客户端程序集使用定义了代理类型的代码文件（以及应用程序配置文件中的设置）去与远程类型进行通信。

本章的结尾部分提及了WCF编程工具SvcConfigEditor.exe（使用它可以修改*.config文件）、WcfTestClient.exe应用程序（快速测试WCF服务）和各种Visual Studio WCF项目模板。你还学习了一些简化配置，包括默认的终结点和行为。

第 26 章

Windows Workflow Foundation 简介

我们知道，.NET平台支持Windows Workflow Foundation（WF）编程模型。该API可以对给定的.NET程序内部所使用的*工作流*（workflow，业务流程）进行建模、配置、监视和执行。默认情况下，工作流的建模使用一种声明式的基于XML的语法XAML。在XAML中，工作流所使用的数据被视为一等公民。

如果你是WF新手，那么本章首先定义了什么是业务流程，然后描述业务流程是如何与WF API相关联的。在此过程中，你将了解到WF活动（activity）的概念、工作流的公共类型以及各种项目模板和编程工具。在讲解了基础知识之后，我们将编写几个示例程序，说明如何利用WF编程模型来建立在WF运行时引擎监视之下执行的业务流程。

说明 WF API的整个内容不可能在这么一个介绍性的章节里全部涵盖。如果你需要了解更深入的内容，请参考Bayer White所著的*Pro WF 4.5*（Apress, 2012）。

26.1 定义业务流程

任何现实世界的应用程序都必须能为各种*业务流程*建模。简单地说，一个业务流程是对逻辑上作为一个整体协同工作的多个任务在概念上进行组合。例如，构建一个让用户在线购买汽车的应用程序。在用户提交订单后，很多活动开始运转。我们也许首先执行信用检查。如果用户通过了信用验证，就执行一个数据库事务，从Inventory表中删除一条记录，在Orders表中添加一条记录，并更新用户的账户信息。在数据库事务完成之后，我们也许还需要给买方发送一个确认电子邮件，然后调用一个远程服务向汽车代理商下订单。总的来说，所有这些任务代表了一个单独的业务流程。

一直以来，业务流程建模都是程序员需要应对的另一个细节，经常需要通过自定义的代码来确保业务流程能够正确地建模，并且确保其在应用程序内部能够正确地执行。例如，你可能需要编写其他的代码来处理流程中出现的错误、跟踪和日志记录支持（用来查看指定业务流程进行的情况）、持久支持（用来保存长久运行的流程的状态）以及其他一些东西。你大概有过亲身体验，构建这种基础设施需要花费很长时间和很多手工劳动。

假设一个开发团队已经为他们的应用程序打造了一个自定义的业务流程框架，但是他们的工作还

没有结束。简单地说，团队中除了程序员以外，还有其他人员也需要了解业务流程，而向他们解释原生的C#代码库并不是一件容易的事情。事实上，业务方面的专家、经理、销售人员和美工设计团队的成员并不懂代码语言。鉴于这个事实，作为程序员，我们需要利用其他的建模工具（例如，使用微软的Visio，或直接贴在办公室的公告栏上），使用非技术性的中性术语以图形的形式来表示流程。这里很明显的问题是，我们需要保持两个实体的一致性，即如果我们改动了代码，就需要更新图表；假如我们改动了图表，代码也要随之更新。

此外，当使用100%代码的方式来构建一个复杂的软件应用程序时，在代码库里将看不出应用程序内部流程的痕迹。例如，一个典型的.NET程序可能由成百上千个自定义类型组成（更不用说所使用的基础类库的种种类型了）。尽管程序员知道对象间是如何调用的，但是代码本身跟一个解释整个活动序列的活文档相比，还是差得很远。开发团队也许会创建外部文档和工作流程图，但是这同样还是会遇到一个流程多种表现形式的问题。

WF的作用

实质上，WF API允许程序员使用预制的活动集声明式地设计业务流程。这样，并不需要创建自定义的程序集来表示一个给定的业务活动和所需的基础设施，我们就可以利用Visual Studio的WF设计器，在设计阶段创建我们的业务流程了。这样，我们可以通过WF构建一个业务流程的骨架，然后再用代码来实现。

WF API编程可用单个实体来表示整个业务流程以及定义这个流程的代码。因为单个WF文档不仅为业务流程提供了一个友好的视觉表示，它还可以用来代表驱动流程的代码，所以我们不再需要担心多个文档不同步的问题了。更棒的是，这个WF文档能清楚地说明流程本身。加上一点点的指导，即使那些最不了解技术的团队成员也应该能理解WF设计器所建立的模型。

26.2　构建简单的工作流

当构建一个启用工作流的应用程序时，你肯定会注意到它与构建传统.NET应用程序有所不同。例如，本书到目前为止，每个代码例子都从创建一个新的项目工作空间开始（如最常见的控制台应用程序），然后编写代码来表示详细的程序。一个WF应用程序也由自定义代码组成，但你同时还要把业务流程本身直接构建到程序集中去。

WF另一个与其他类型的.NET应用程序不同的方面是，大多数工作流模型都使用基于XML的语法XAML来声明式地构建。大多数情况下，你不需要直接编写这些标记，因为在你使用WF设计器工具时，Visual Studio IDE会自动完成这些工作。这是WF API较以往的版本一个很大的改变，它有利于用标准的C#代码建立工作流模型。

说明　注意，WF所使用的XAML方言与WPF使用的XAML方言是不同的。你将在第27章学习WPF XAML的语法和符号。与WF XAML不同，编辑设计器生成的WPF XAML是很常见的事情。

为了深入学习工作流，我们打开Visual Studio，在New Project对话框中选择一个Workflow Console Application项目，并将其命名为FirstWorkflowExampleApp（如图26-1所示）。

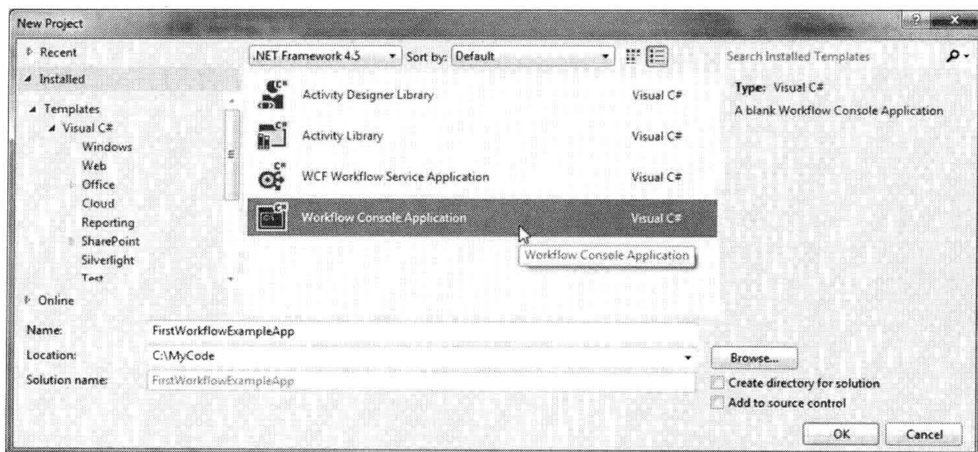


图26-1　创建一个基于控制台的工作流应用程序

图26-2演示了Visual Studio生成的初始的工作流图表。如你所见，这时设计器中除了一个提醒你放置活动的信息外什么也没有。



图26-2　工作流设计器是活动的容器，活动为业务流程建模

在这个简单的测试工作流中，打开Visual Studio的Toolbox，找到Primitives节下的`WriteLine`活动（如图26-3所示）。

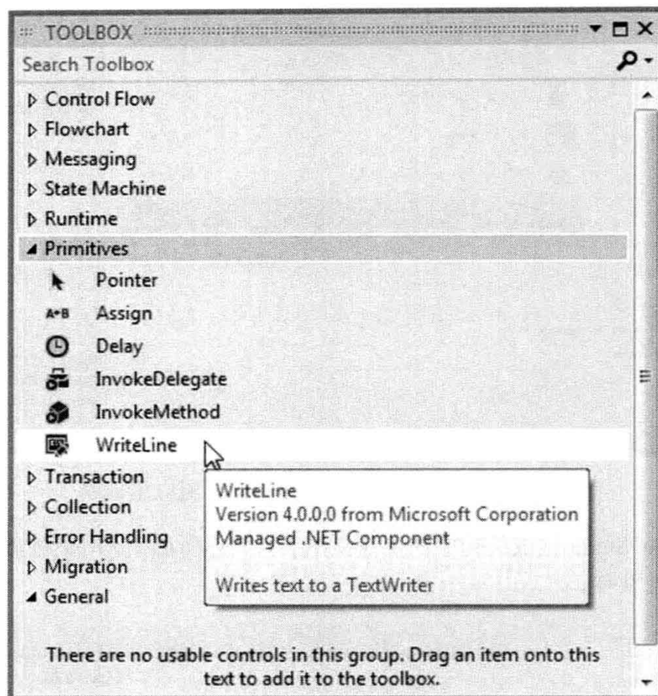


图26-3　Toolbox显示了WF中所有的默认活动

找到该活动之后，将其拖曳到设计器中（写着“Drop activity here”的区域），并在Text编辑框中输入用双引号括起的字符串信息，如图26-4所示。

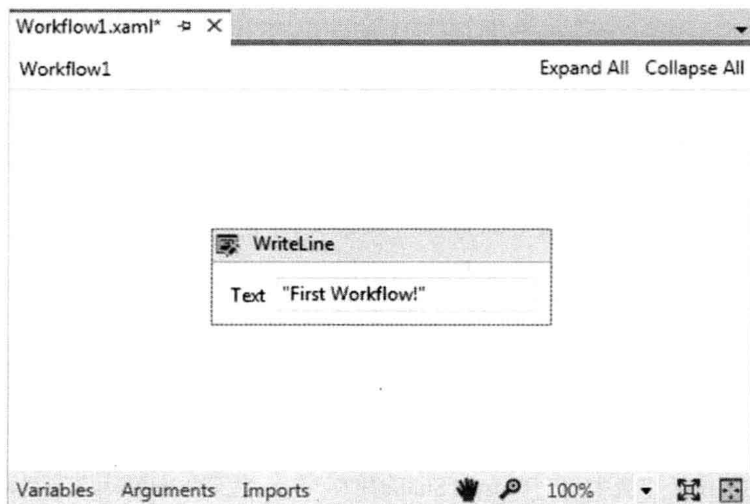


图26-4　WriteLine活动将向TextWriter（本例中的控制台）显示文本

要知道WF远不止一个可以对业务流程中的活动进行建模的设计器。在构建WF图表时，可以使用代码对标记进行扩展，来表示流程在运行时的行为。事实上，你完全可以避开XAML，而仅仅使用C#编写工作流。但这么做的话，就又回到那个老问题上，即拥有一堆非技术人员看不懂的代码体。无论如何，运行应用程序，将会在控制台窗口看到如下所示的消息：

```
First Workflow!
Press any key to continue . . .
```

非常好。但是，工作流如何启动？如何确保控制台应用程序在工作流完成之前一直运行？要回答这些问题，需要了解工作流的运行时引擎。

26.3 Workflow 运行时

接下来要了解的是，WF API还包含一个运行时引擎，用来加载、执行、卸载以及用其他方式操作定义好的工作流。WF运行时引擎可以寄宿在任何.NET应用程序域中。但要注意的是，一个单独的应用程序域只能包含一个正在运行的WF引擎实例。

第17章已经介绍过，AppDomain是Windows进程的分区，后者是.NET应用程序和任何外部代码库的宿主。因此，WF引擎可以内嵌到简单的控制台程序、GUI桌面应用（Windows Forms或WPF），或暴露于WCF（Windows Communication Foundation）服务中。

26

说明 如果你希望构建一个在内部使用工作流的WCF服务（见第25章），WCF Workflow Service Application项目模板是一个不错的起点。

如果你正在建模的业务流程需要使用各种各样的系统，你也可以选择在C# Class Library项目（Workflow Activity Library项目）中编写WF。这样，新的应用程序通过简单地引用*.dll就可以重用这些预定义的业务流程集合。如果你想避免多次创建同样的工作流，这样做显然很有帮助。

26.3.1 使用WorkflowInvoker承载工作流

WF运行时的宿主进程使用一些不同的技术与运行时进行交互。其中最简单的方式是使用System.Activities命名空间下的WorkflowInvoker类。它仅用一行代码就可以启动一个工作流。打开当前Workflow Console Application项目的Program.cs文件，可以看到如下的Main()方法：

```
static void Main(string[] args)
{
  // 创建并缓存工作流定义
  Activity workflow1 = new Workflow1();
  WorkflowInvoker.Invoke(workflow1);
}
```

当你只想简单地启动工作流而不希望进一步监控时，使用WorkflowInvoker是十分有用的。Invoke()方法将以同步阻塞方式（synchronous blocking manner）执行工作流。在整个工作流结束或突然中断前，

调用线程都将保持阻塞状态。由于Invoke()方法是同步调用，这确保了整个工作流在Main()终止前能够完成。事实上，在WorkflowInvoker.Invoke()方法之后添加的任何代码，都将在工作流完成（或更糟的情况，突然中断）之后才能执行：

```
static void Main(string[] args)
{
  // 创建并缓存工作流定义
  Activity workflow1 = new Workflow1();
  WorkflowInvoker.Invoke(workflow1);

  Console.WriteLine("Thanks for playing");
}
```

1. 使用WorkflowInvoker向工作流传递参数

当宿主进程开启一个工作流时，传递一些自定义启动参数是很常见的情况。例如，假设你希望程序的用户指定显示在WriteLine活动中的消息，而不是像现在这样将文本消息硬编码到活动中。在普通的C#代码中，你需要创建一个自定义的构造函数来接收这样的参数。但是，工作流只能用默认的构造函数创建！此外，大多数工作流只使用XAML定义，而不是程序代码。

事实上，Invoke()方法有很多重载，其中一个允许你向工作流传递启动参数。这些参数保存在一个Dictionary<string, object>变量中，它包含的名称/值对可用来设置工作流中同名（同类型）的参数变量。

2. 使用工作流设计器定义参数

可以使用工作流设计器来定义参数并获取字典数据。在Solution Explorer中，右击Workflow1.xaml选择View Designer。注意在设计器底部有一个名为Arguments的按钮。单击该按钮，在弹出的UI中，添加一个string类型的输入参数MessageToShow（不需要为该参数指定默认值）。同样，通过在Visual Studio 的Properties窗口中重置WriteLine活动的Text属性，来删除WriteLine活动中的初始消息。图26-5显示了最终的结果。

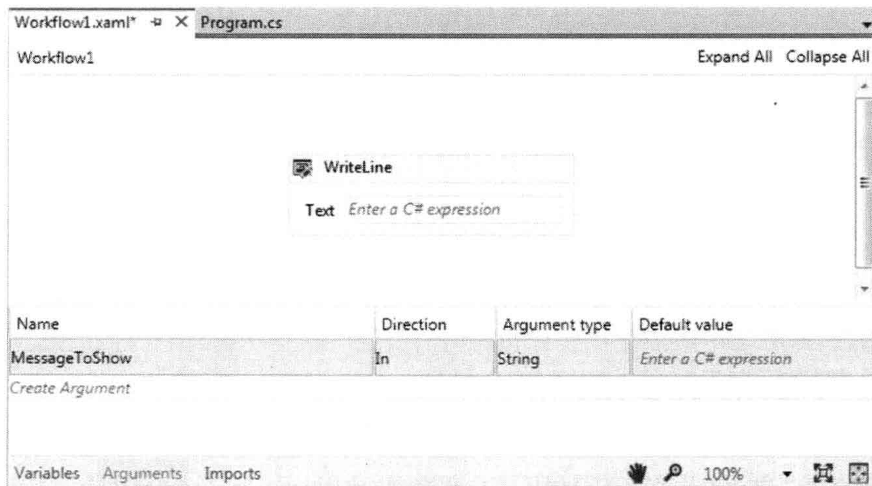


图26-5　工作流参数可用来接收宿主提供的参数

现在，在WriteLine活动的Text属性中，可以输入MessageToShow作为赋值表达式。当你输入这些字符时，可以看到智能感知（如图26-6所示）。

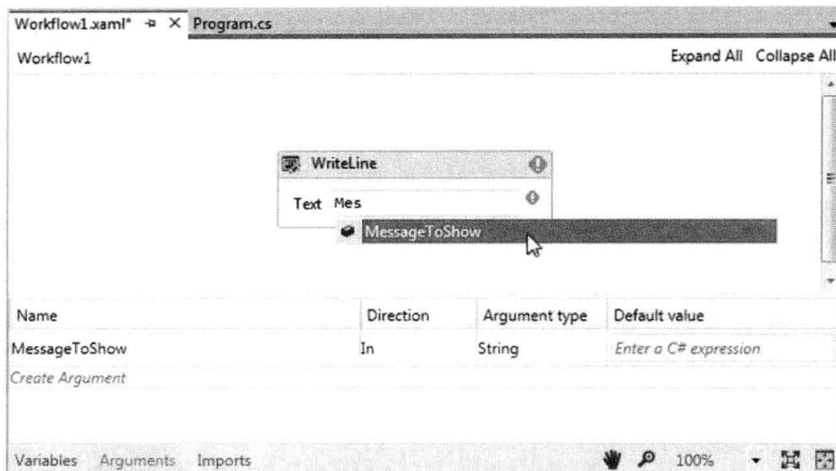


图26-6 使用自定义参数作为活动的输入

现在已经有了正确的基础设施，考虑下面对Program类的Main()方法所做的修改。注意你需要在Program.cs文件中引入System.Collections.Generic命名空间，来声明Dictionary<>变量。


```
static void Main(string[] args)
{
  Console.WriteLine("***** Welcome to this amazing WF application *****");

  // 获取用户输入的数据，传递给工作流
  Console.Write("Please enter the data to pass the workflow: ");
  string wfData = Console.ReadLine();

  // 将数据存入字典中
  Dictionary<string, object> wfArgs = new Dictionary<string,object>();
  wfArgs.Add("MessageToShow", wfData);

  // 传递给工作流
  Activity workflow1 = new Workflow1();
  WorkflowInvoker.Invoke(workflow1, wfArgs);

  Console.WriteLine("Thanks for playing");
}
```

再次强调，Dictionary<>变量中每个成员的字符串值必须与工作流中相应的参数变量名一致。运行修改后的程序，输出结果如下所示：

```
***** Welcome to this amazing WF application *****
Please enter the data to pass the workflow: Hello Mr. Workflow!
Hello Mr. Workflow!
Thanks for playing
Press any key to continue ...
```

除了Invoke()方法，WorkflowInvoker中其他有趣的成员是BeginInvoke()和EndInvoke()，它们使用.NET异步委托模式（如第19章所示）在另一个线程中启动工作流。如果你希望对WF运行时操作工作流有更多地控制，可以使用WorkflowApplication类。

26.3.2 使用WorkflowApplication承载工作流

如果要保存或加载一个使用WF持久化服务长期运行的工作流，可以使用WorkflowApplication。它具有接收工作流实例生命周期中触发的各种事件的通知、WF“书签”等其他高级特性。在这种情况下，需要调用WorKflowApplication的Run()方法。

调用Run()方法时，将从CLR线程池中取出一个后台线程。因此，如果你不添加额外的支持来保证主线程等待辅助线程执行完毕的话，那么工作流实例可能没有机会完成其工作。

要想让调用线程等待足够长的时间以使后台线程能够完成其工作，一种方法是使用System.Threading命名空间下的AutoResetEvent对象。以下是修改后的示例，使用WorkflowApplication替代了WorkflowInvoker：

```
static void Main(string[] args)
{
  Console.WriteLine("***** Welcome to this amazing WF application *****");

  // 获取用户输入的数据，传递给工作流
  Console.Write("Please enter the data to pass the workflow: ");
  string wfData = Console.ReadLine();

  // 将数据存入字典中
  Dictionary<string, object> wfArgs = new Dictionary<string,object>();
  wfArgs.Add("MessageToShow", wfData);

  // 通知主线程进行等待
  AutoResetEvent waitHandle = new AutoResetEvent(false);

  // 传递给工作流
  WorkflowApplication app = new WorkflowApplication(new Workflow1(), wfArgs);

  // 将事件与app挂钩。当工作流结束时，通知其他线程，并打印一条信息
  app.Completed = (completedArgs) => {
    waitHandle.Set();
    Console.WriteLine("The workflow is done!");
  };

  // 开启工作流
  app.Run();

  // 在工作流结束之前一直等待
  waitHandle.WaitOne();

  Console.WriteLine("Thanks for playing");
}
```

输出结果与上例类似：

```
***** Welcome to this amazing WF application *****
Please enter the data to pass the workflow: Hey again!
```

```
Hey again!
The workflow is done!
Thanks for playing
Press any key to continue . . .
```

使用WorkflowApplication的好处是，可以将工作流与事件挂钩（如在本例中间接使用Completed属性），并且可以使用更复杂的服务（如持久化、书签等）。

说明 在我们介绍WF的时候，不会深入这些运行时服务的细节。如果对Windows Workflow Foundation运行时环境下的这些运行时行为和服务感兴趣，可以参阅.NET Framework 4.5 SDK文档。

26.3.3 第一个工作流示例回顾

尽管这个示例有点简单，但你确实学到了一些有趣（且有用）的任务。首先，我们介绍了可以向工作流传递一个Dictionary对象，它包含的名称/值对将传递给工作流中具有相同名称的参数。当你需要获得用来处理活动的用户输入时（如客户的ID号、SSN、私人医生的名字等），这种方法相当有用。

我们还介绍了.NET 工作流是使用XAML（基于XML的语法）以一种声明式的方式来定义的。使用XAML，可以指定工作流包含哪些活动。在运行时，这些数据用来创建正确的在内存中的对象模型。最后，我们介绍了使用WorkflowInvoker和WorkflowApplication这两种不同的方法来启动工作流。

26

源代码 FirstWorkflowExampleApp项目的源代码位于Chapter 26子目录下。

26.4 检查 Workflow 中的活动

WF的目的是以声明的方式对业务流程建模，并由WF运行时引擎执行。在WF的世界里，业务流程是由多个活动组成的。简单地说，在整个流程中，一个WF活动就是一个原子的"步骤"。当你创建一个工作流应用程序时，你会发现Toolbox中内嵌了很多按种类划分的活动，它们以图标的形式表示。

这些随手可用的活动可用来对业务流程建模。Toolbox中的每个活动都对应System.Activities.dll程序集中一个实际的类（大多数都位于System.Activities.Statements命名空间下）。你将在本章的教程中使用一些内嵌的活动。我们也将在此对大多数默认的活动进行概述。和往常一样，详细的内容请参考.NET Framework SDK文档。

26.4.1 控制流活动

工具箱中的第一种活动允许你在较大的工作流中表示循环和判定。它们的用途应该很容易理解，因为我们每天都用C#代码做着同样的事情。在表26-1中，注意某些控制流活动在后台使用了Task Parallel Library（任务并行库，参见第19章），允许并行地处理活动。

表26-1 WF中的控制流活动

活　　动	含　　义
DoWile	一个循环活动，该活动至少执行内含的活动一次，直到条件不为true
ForEach<T>	将ForEach<T>.Values集合中的各个活动分别执行一次
If	建立一个If-Then-Else条件模型
Parallel	以异步的方式同时执行的所有子活动的活动
ParallelForEach<T>	枚举集合中的元素，并以并行的方式执行各个元素
Pick	提供基于事件的控制流建模
PickBranch	父Pick活动内可能的执行路径
Sequence	执行一组子活动序列
Switch<T>	基于给定表达式（其类型在此对象的类型参数中指定）的值，从多个活动中选择一个执行
While	在条件的值为true时执行包含的工作流元素

26.4.2 流程图活动

接下来介绍的是流程图活动，这实际上是非常重要的，因为它常常是我们放置到WF设计器上的第一个项。这种类型的工作流允许你使用众所周知的流程图模型来构建工作流。工作流的执行基于许多分支路径，选择哪条分支路径取决于一些内部条件的真或假。表26-2列出了该活动集合的所有成员。

表26-2 WF中的流程图活动

活　　动	含　　义
Flowchart	使用熟悉的流程图范例建立工作流的模型。它常常是放到设计器中的第一个活动
FlowDecision	一个节点，提供建立有两种结果的条件节点模型的能力
FlowSwitch<T>	一个节点，可建立开关结构的模型，该结构有一个表达式并且每个匹配项都有一个结果

26.4.3 消息传递活动

工作流可以使用消息传递活动轻松地调用外部XML Web服务或WCF服务的成员，并且也可以接收外部服务的通知。由于这些活动与WCF开发息息相关，因此存放于专门的程序集System.ServiceModel.Activities.dll中。该库中的核心活动如表26-3所示。

表26-3 WF中常见的消息传递活动

活　　动	含　　义
CorrelationScope	管理子消息传递活动
InitializeCorrelation	初始化关联（correlation）而不发送或接受消息
Receive	从WCF服务接收消息
Send	将消息发送到WCF服务
SendAndReceiveRelply	将消息发送到WCF服务并获取返回值
TransactedReceiveScope	允许你将事务流入工作流或调度创建的服务器事务

最常见的消息传递活动是Send和Receive，可以用来与外部XML Web服务和WCF服务进行通信。

26.4.4 状态机活动

在.NET 4.5下，WF API得到了更新，新增了一些可以对基于状态机的工作流建模的活动。简单来说，状态机允许我们定义一种工作流，这种工作流可以在某个给定的时间点定义任意数量的状态，并且这些状态之间可以有效地转换。

关于状态机一个众所周知的例子是汽水自动售货机。在任意给定的时间点，该“机器”都处于一种状态，如“等待付款”、“配制汽水”、“退还付款”、“找零”、“所选售空”等。这些状态之间可以有效地转换。例如，如果售货机处于“所选售空”状态，那么可以有效地转换为“退还付款”或“配制汽水”（假设用户选择了另一种汽水）。要构建这样一个工作流，.NET 4.5引入了StateMachine、State和FinalState活动。

26.4.5 运行时活动与基元活动

工具箱中下面的两种活动是运行时活动和基元活动。运行时活动可用来构建调用了工作流运行时（如Persist和TerminateWorkflow）的工作流，基元活动可用来构建执行常用操作（如向输出流推送文本，或调用.NET对象中的方法）的工作流，表26-4展示了该类别中常见的活动。

表26-4 WF的运行时活动和基元活动

活 动	含 义
Persist	使用WF持久化服务将工作流实例的状态保存到数据库中的请求
TerminateWorkflow	终止正在运行的工作流实例，在宿主中触发WorkflowApplication.Completed事件，并报告错误信息。工作流终止后将无法恢复
Assign	使用工作流设计器中定义的值来设置活动的属性
Delay	在一个固定的时间点强制停止一个工作流
InvokeMethod	调用指定对象或类型的方法
WriteLine	将指定的字符串写入指定的TextWriter派生类型。默认为标准输出流（即控制台），但也可以配置为其他流，如FileStream

InvokeMethod也许是这些活动中最有趣也最有用的一个，因为它支持以声明的方式调用.NET类中的方法。你还可以配置InvokeMethod来保存所调用方法的返回值。当你想终止一个工作流时，TerminateWorkflow也是很有帮助的。如果工作流实例命中该活动，将触发Completed事件，宿主程序将捕捉到该事件，就如同我们在前面的示例中所做的那样。

26.4.6 事务活动

在构建工作流时，你也许会希望让一组活动按原子的方式来工作，即作为一个集体，要么都成功，要么都失败。尽管这些活动不直接操作关系型数据库，但核心活动允许你向工作流中添加事务范围，如表26-5所示。

表26-5　WF的事务活动

活　动	含　义
CancellationScope	将取消逻辑与执行的主路径相关联
CompensableActivity	支持对子活动进行补偿的活动
TransactionScope	划分事务边界的活动

26.4.7　集合活动和错误处理活动

本章介绍的最后两类活动可以操作泛型集合和响应运行时异常。当你需要在XAML中操作表示业务数据（如采购订单、医疗信息对象或订单跟踪）的对象时，可以使用强大的集合活动。而错误活动支持在工作流中添加try/catch/throw逻辑。表26-6列出了这些活动。

表26-6　WF的集合活动和错误处理活动

活　动	含　义
AddToCollection<T>	向指定的集合中添加项
ClearCollection<T>	清除指定集合中的所有项
ExistsInCollection<T>	指示给定项是否存在于给定集合中
RemoveFromCollection<T>	从指定的集合中移除项
Rethrow	从Catch活动内抛出一个以前抛出过的异常
Throw	抛出异常
TryCatch	包含由工作流运行时在异常处理块中执行的工作流元素

好了，我们已经介绍了很多默认的活动，可以使用它们来构建一些有趣的工作流了。接下来我们将学习两个关键的活动Flowchart和Sequence，它们通常被认为是工作流的根本。

26.5　构建流程图工作流

在第一个示例中，我们将一个简单的WriteLine活动直接拖曳到工作流设计器上。尽管Visual Studio工具箱中的任何活动都可以作为第一项放置到设计器中，但只有一小部分可以包含子活动（表示一组相关活动的集合）。在构建新的工作流时，第一个放置到设计器中的项常常是Flowchart或Sequence活动。

这两个内嵌的活动都可以包含任意数量的内部子活动（可以为其他的Flowchart或Sequence活动），来表示整个业务流程。我们创建一个全新的Workflow Console Application命名为EnumerateMachine-DataWF，并将初始的*.xaml文件改名为MachineInfoWF.xaml。

现在，从Toolbox中的Flowchart节下，将一个Flowchart活动拖曳到设计器中。接下来，在Properties窗口中将DisplayName属性更改为更容易记住的名字，如Show Machine Data Flowchart（我敢肯定你在猜想DisplayName属性控制了该项在设计器中的命名）。这时，工作流设计器将如图26-7所示。

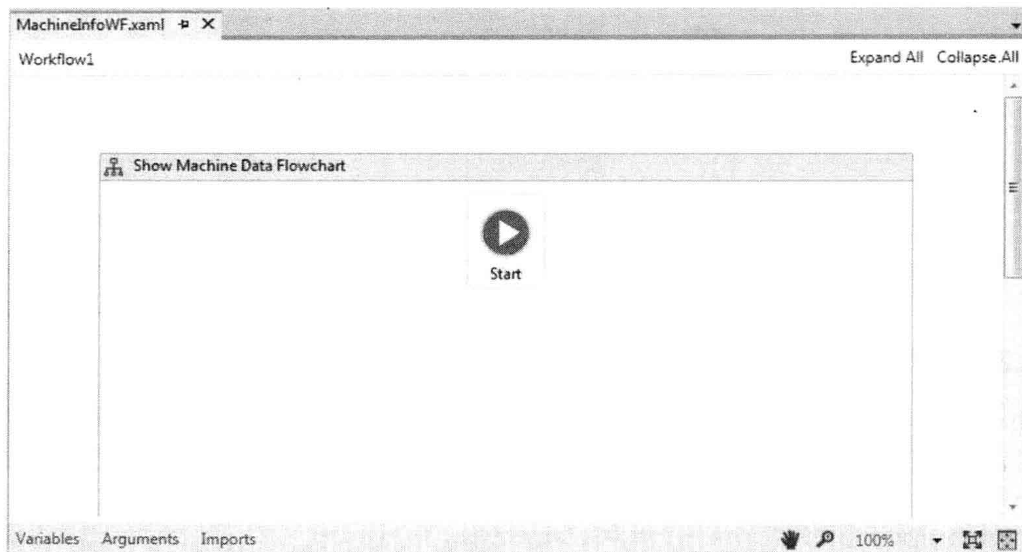


图26-7 初始的Flowchart活动

注意，在Flowchart活动的右下角有一个缩放处理的图标，可以用来增大或缩小流程图设计器空间的尺寸。随着活动的增多，你将需要扩大这个尺寸。

26.5.1 在流程图中连接活动

Start图标表示该流程图活动的入口点，在本例中它是我们放置的整个工作流中的第一个活动，并且将在使用WorkflowInvoker或WorkflowApplication类执行工作流时被触发。你可以将这个图标拖动到设计器中的任何位置，我建议将它移动到左上角，以便留出更多的空间。

我们的目标是通过连接多个活动将它们组装在一起，在流程中要用到FlowDecision活动。我们将WriteLine活动拖曳到设计器中，将DisplayName改为Greet User。现在，如果你将鼠标悬停在Start图标上，会发现四个方向上都出现了对接标记。单击并按住对接标记，然后拖动到WriteLine活动，这样会在这两项中出现一条连接线，这意味着工作流执行的第一个活动是Greet User。

现在，与本章第一个示例相同，我们添加一个名为UserName的无默认值的字符串类型的工作流参数（使用Argument按钮）。该参数将通过自定义的Dictionary<>对象动态地传入。最后，将WriteLine活动的Text属性设置为如下的代码语句：

```
"Hello" + UserName
```

在设计器中再添加一个WriteLine活动，并连接到前一个。在Text属性中硬编码字符串值"Do you want me to list all machine drives?"，并将DisplayName属性改为Ask User。图26-8显示了当前工作流活动的连接情况。

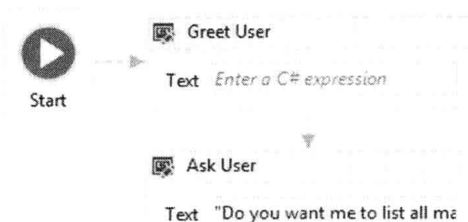


图26-8　流程图工作流将活动连接到一起

26.5.2　使用InvokeMethod活动

由于大多数工作流都使用XAML以声明方式定义，因此我们可以很好地使用InvokeMethod活动，在工作流中不同的位置调用真实对象的方法。将该活动拖曳到设计器中，将DisplayName属性改为Get Y or N，并与Ask User WrilteLine活动进行连接。

InvokeMethod活动要配置的第一个属性为TargetType，它表示类的名称，我们要调用的静态成员就定义在该类中。使用InvokeMethod活动中TargetType的下拉列表框，选择Browse for Types...选项（如图26-9所示）。



图26-9　指定InvokeMethod的目标类型

在弹出的对话框中，选择mscorlib.dll中的System.Console类（在Type Name编辑域中输入类型名称，对话框将自动找到该类型）。找到System.Console类之后，单击OK按钮。

现在，使用设计器中的InvokeMethod活动，将MethodName属性设置为ReadLine。在工作流执行到这一步时，InvokeMethod活动将调用Console.ReadLine()方法。

我们知道，Console.ReadLine()将返回回车键按下之前通过键盘输入的字符串，我们需要一种方法来得到这个返回值。接下来就将介绍这个方法。

26.5.3 定义工作流变量

在XAML中定义变量与定义参数没什么区别，因此可以直接在设计器中进行定义（使用Variables按钮）。所不同的是，参数用来获取宿主程序中传入的数据，而变量只是工作流中影响运行时行为的数据点。

单击设计器中的Variables按钮，添加一个名为YesOrNo的string变量。注意，如果工作流中包含多个父容器（例如，一个Flowchart包含另一个Sequence），可以选择变量的作用域。在此，我们只选择Flowchart根（如图26-10所示）。

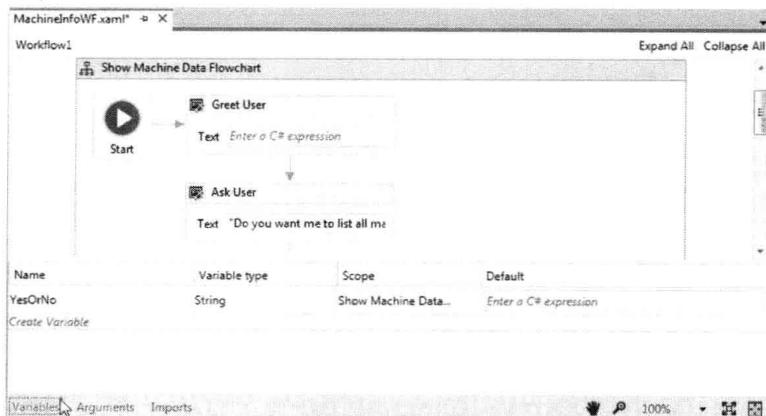


图26-10 定义工作流变量

接下来，在工作流设计器中选择InvokeMethod活动，使用Visual Studio中的Properties窗口，将Result属性设置为刚刚定义的变量（如图26-11所示）。

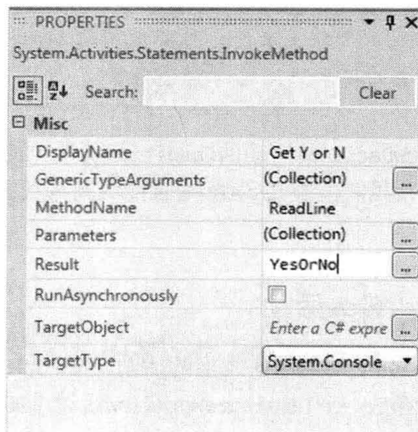


图26-11 InvokeMethod的完整配置

既然我们可以通过调用外部方法获取数据，那么我们就可以在流程图中使用FlowDecision活动来做运行时决策。

26.5.4　使用FlowDecision活动

FlowDesicion活动用来提供两种可能的行为，它基于布尔变量或返回布尔值的语句的真假，来决定执行哪个行为。将该活动拖曳到设计器，与InvokeMethod活动相连接（如图26-12所示）。

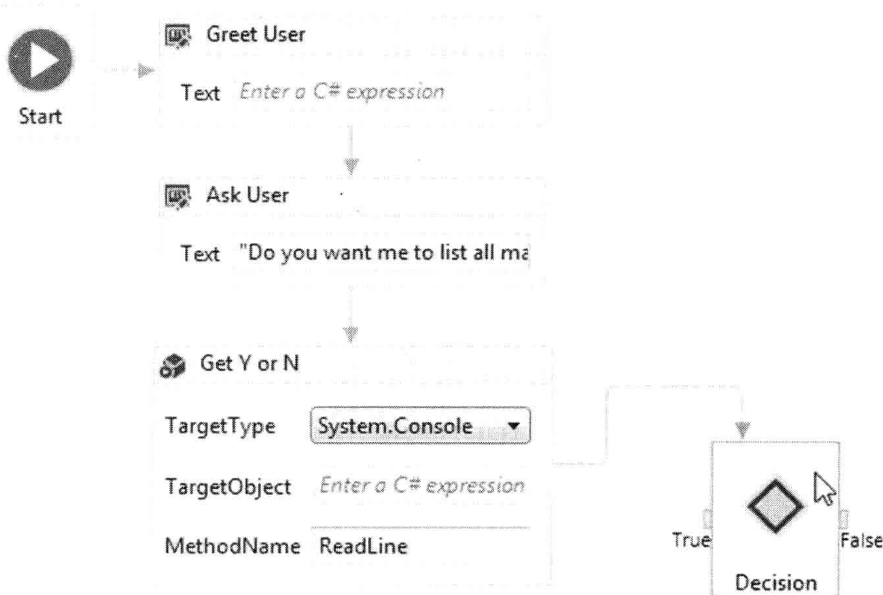


图26-12　FlowDecision可以产生两个方向的分支

说明　如果你需要在流程图中响应多个分支条件，可以使用FlowSwitch<T>活动。它支持定义多个路径，根据定义的工作流变量的值来决定选择哪条路径。

用下面的代码语句来设置FlowDecision活动的Condition属性（使用Properties窗口），你可以直接在编辑器中输入这些代码（这里，我们检查YesOrNo变量的大写形式是否为"Y"）：

```
YesOrNo.ToUpper() == "Y"
```

26.5.5　使用TerminateWorkflow活动

现在我们需要构建FlowDecision活动两端将发生的活动。在“false”端，连接一个WriteLine活动，输出所选择的硬编码消息，紧接着是一个TerminateWorkflow活动（如图26-13所示）。

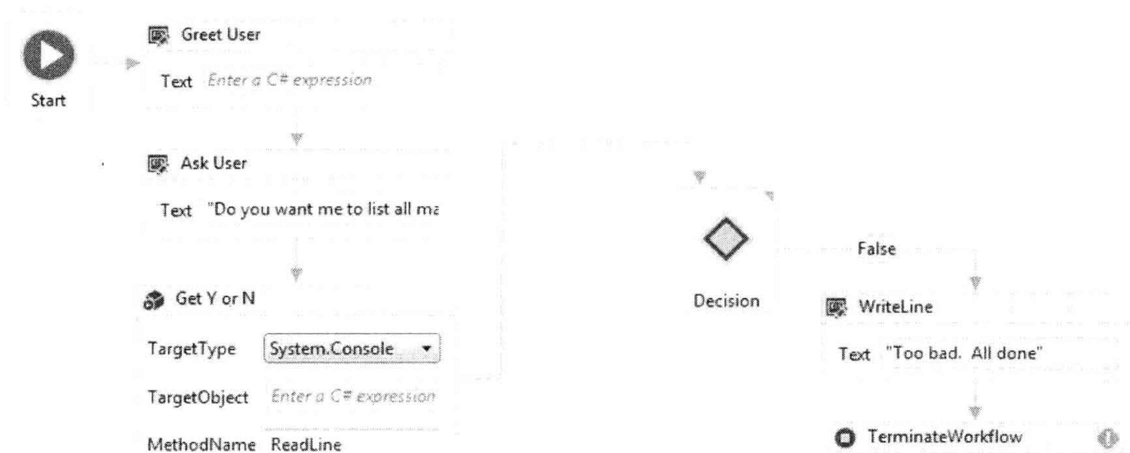


图26-13 “false”分支

严格地说，没有必要使用TerminateWorkflow活动，因为当工作流到达false分支的末端时将会简单地结束。但是，使用该活动类型可以向工作流宿主抛出一个异常，通知终止的原因。你可以在Properties窗口中配置该异常。

假设在设计器中选择了TerminateWorkflow活动，在Properties窗口中单击Exception属性的椭圆形按钮。这将打开一个编辑器，可以像在代码中那样来抛出异常（如图26-14所示）。

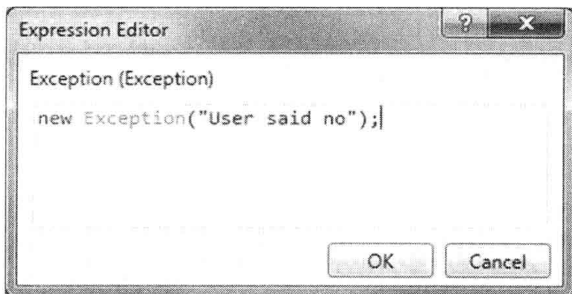


图26-14 配置TerminateWorkflow活动发生时抛出的异常

最后，将该活动的Reason属性设置为“Yes Or No was flase”。

26.5.6 构建“true”条件

要构建FlowDecision的“true”条件，先连接一个WriteLine活动，显示硬编码字符串，表示用户同意继续执行。然后，连接一个新的InvokeMethod活动，调用System.Environment类的GetLogicalDrives()方法。将TargetType属性设置为System.Environment，MethodName属性设置为GetLogicalDrives（如图26-15所示）。

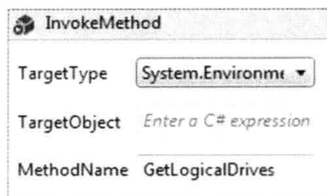


图26-15　配置InvokeMethod活动

接下来，添加一个工作流级别的变量（使用工作流设计器中的Variables按钮）DriveNames，其类型为string[]。要指定string数组，需要在Variable Type的下拉列表中选择Array of [T]，然后在弹出的对话框中选择String。最后，在设计器上选中InvokeMethod活动，在Properties窗口中将该InvokeMethod活动的Result属性设置为DriveNames变量。

26.5.7　使用 ForEach<T>活动

工作流接下来的部分就是要在控制台窗口打印每个驱动器的名称，这意味着我们需要循环DriveNames变量（string数组对象）中的数据。WF中的ForEach<T>活动等价于C#中的foreach关键字，它们的配置方式也十分类似（至少在概念上是这样）。

将一个ForEach<T>活动拖曳到设计器中，并与之前的InvokeMethod活动相连接。稍后我们再配置ForEach<T>活动。此时我们先完成true条件分支，将最后一个WriteLine活动放到设计器上。此时流程图的整体结构如图26-16所示。

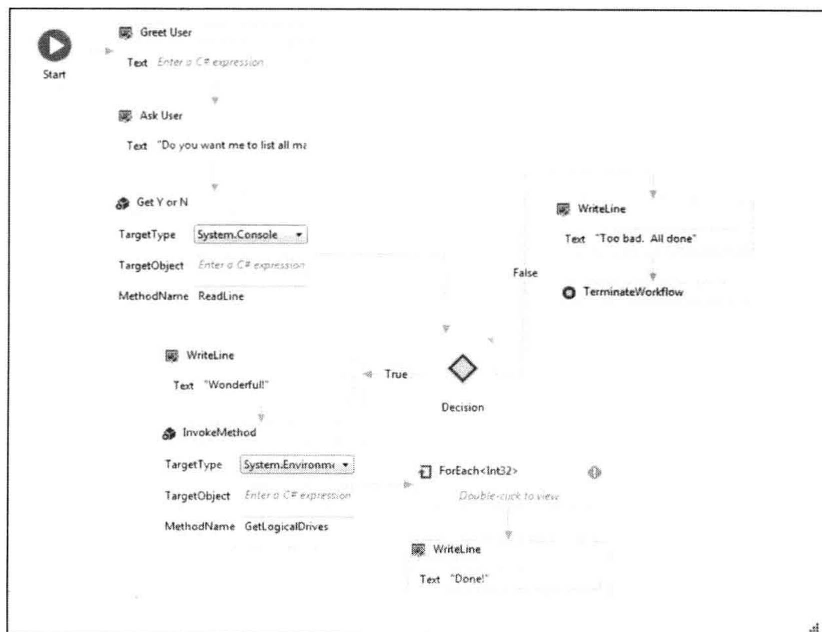


图26-16　完整的工作流

为了避免当前设计器中的错误，需要完成ForEach<T>活动的配置。首先，使用Properties窗口泛型的类型实参指定为String类型。Values属性表示数据的来源，这里应为DriveNames变量（如图26-17所示）。

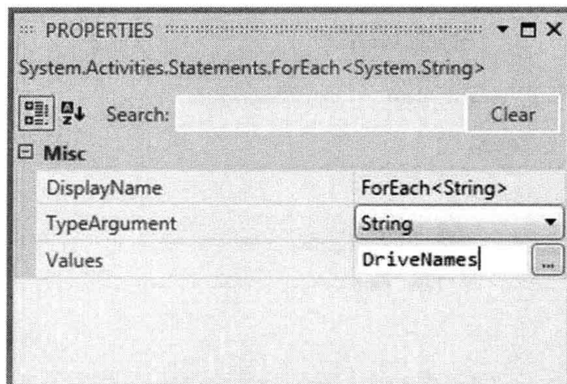


图26-17 设置ForEach枚举的类型

这个特殊的活动需要进一步的编辑，在设计器中双击该活动，打开一个该活动专用的迷你设计器。并非所有的WF活动都可以双击，但活动本身会告诉你是否可以这么做（显示“Double-click to view”的字样）。双击ForEach<String>活动，添加一个WriteLine活动，用来打印DriveNames返回值中每个string的值（如图26-18所示）。

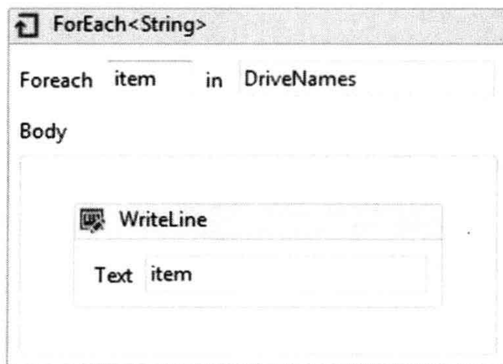


图26-18 ForEach<String>活动的最终配置

说明 可以在ForEach<T>的迷你设计器中添加任意多个活动。这些活动将在循环的每次迭代中执行。

配置好ForEach<T>的“子活动”之后，可以使用工作流设计器左上角的链接返回到工作流的最顶端（在你深入一组活动中时，将会经常使用这些链接；请看图26-19中的鼠标位置）。



图26-19　通过工作流设计器中的“链接”可以返回活动的顶端

26.5.8　完成应用程序

示例就要完成了。你还需要做的是更新Program类的Main()方法，捕获用户输入“NO”时触发的异常(触发Exception对象)。按如下的代码进行更新(确保代码文件引入了System.Collections.Generic命名空间)。

```
static void Main(string[] args)
{
  try
  {
    Dictionary<string, object> wfArgs = new Dictionary<string, object>();
wfArgs.Add("UserName", "Mel");
Activity workflow1 = new Workflow1();
    WorkflowInvoker.Invoke(workflow1, wfArgs);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    Console.WriteLine(ex.Data["Reason"]);
  }
}
```

注意，异常的“Reason”可以通过System.Exception的Data属性获取。运行应用程序，并在询问是否枚举驱动器时输入“Y”，输出结果如下所示：

```
Hello Andrew
Do you want me to list all machine drives?
y
Wonderful!
C:\
D:\
E:\
F:\
G:\
H:\
I:\
Thanks for using this workflow
```

如果输入“N”（或其他非“Y”或“y”的值），将得到如下结果：

```
Hello Andrew
Do you want me to list all machine drives?
n
Too bad. All done
YesOrNo was false
```

26.5.9 我们做了什么

如果你刚刚接触工作流环境，可能不知道我们使用WF XAML而非纯C#代码编写这些简单的业务逻辑之后到底得到了什么。毕竟，你可以完全绕开Windows Workflow Foundation，而编写如下的C#类：

```
class Program
{
  static void Main(string[] args)
  {
    try
    {
      ExecuteBusinessProcess();
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
      Console.WriteLine(ex.Data["Reason"]);
    }
  }

  private static void ExecuteBusinessProcess()
  {
    string UserName = "Andrew";
    Console.WriteLine("Hello {0}", UserName);
    Console.WriteLine("Do you want me to list all machine drives?");

    string YesOrNo = Console.ReadLine();
    if (YesOrNo.ToUpper() == "Y")
    {
      Console.WriteLine("Wonderful!");
      string[] DriveNames = Environment.GetLogicalDrives();
      foreach (string item in DriveNames)
```

```
        {
          Console.WriteLine(item);
        }
        Console.WriteLine("Thanks for using this workflow");
      }
      else
      {
        Console.WriteLine("K, Bye...");
        Exception ex = new Exception("User Said No!");
        ex.Data["Reason"] = "YesOrNo was false";
      }
    }
  }
```

这段程序的输出结果将和前面基于工作流的XAML的输出结果完全一致。那么，我们为什么还要去捣这些活动呢？首先，并不是所有人都能读懂C#代码。坦诚地说，如果你需要给一屋子的销售人员和非技术管理者解释这个业务逻辑，你是会解释这段C#代码呢，还是向他们展示这个流程图？更重要的是，WF API包含了众多额外的运行时服务，包括将长期运行的工作流保存到数据库、自动跟踪工作流事件，等等（恕不一一列举）。如果要把这些功能复制到新的项目中，WF所需的工作量显然更少。

尽管WF API并不一定是所有.NET程序的正确选择，但对大多数传统业务的应用程序来说，能够以这种方式定义、承载、执行和监控工作流确实是一件幸事。和其他新技术一样，你需要决定它是否对当前项目有用。让我们来看看WF API的另一个示例，这次我们将工作流包装在一个专门的*.dll文件中。

源代码　EnumerateMachineDataWF项目的源代码位于Chapter 26子目录下。

26.6　在专门的 DLL 中构建 Squence 工作流

尽管Workflow Console Application是对WF API进行的伟大尝试，但一个准备投产的工作流将肯定被打包成自定义的.NET *.dll程序集。这样就可以在多个项目中重用以二进制形式存在的工作流。

你可以使用C# Class Library项目建立工作流库，但最简单的方法是使用Activity Library项目，它位于New Project对话框的Workflow节点下。使用该项目类型的好处是，它自动引用了所需的WF程序集，并给出了一个*.xaml文件用于创建初始的程序流。

在本节的工作流中，我们将查询AutoLot数据库，检查Inventory表中是否包含给定品牌和颜色的汽车，并对这一过程进行建模。如果被请求的汽车有现货，则构建一个格式良好的响应，通过输出参数传递给宿主程序。如果没有现货，将生成一个备忘录，发送给销售部门的负责人，请求他们找出一个符合条件的汽车。

26.6.1　定义初始化项目

创建一个新的Activity Library项目CheckInventoryWorkflowLib（如图26-20所示），将初始的Activity1.xaml文件改名为CheckInventory.xaml。

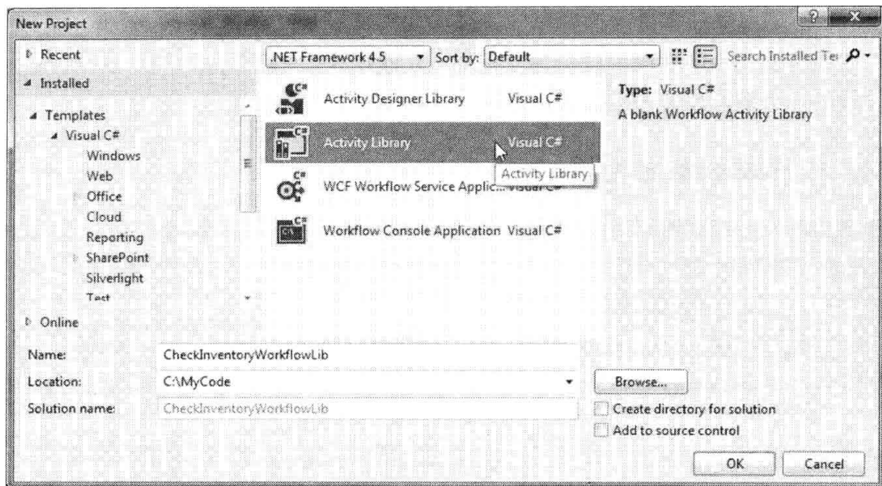


图26-20　创建一个Activity Library项目

不幸的是，当为一个工作流XAML文件重命名时，后台的类将不会被重命名。要解决这个问题，可以在Solution Explorer中右键点击CheckInventory.xaml文件，选择查看代码。修改`<Activity>`根元素以反映真实名称，如下所示：

```
<Activity mc:Ignorable="sap sap2010 sads"
  x:Class="CheckInventoryWorkflowLib.CheckInventory"
  ...
>
```

该工作流将使用一个`Sequence`活动作为主活动，而不是`Flowchart`。将一个`Sequence`活动拖曳到设计器中（可以在Toolbox的Control Flow中找到它），并将`DisplayName`属性改为Look Up Product。图26-21展示了当前的设计器。

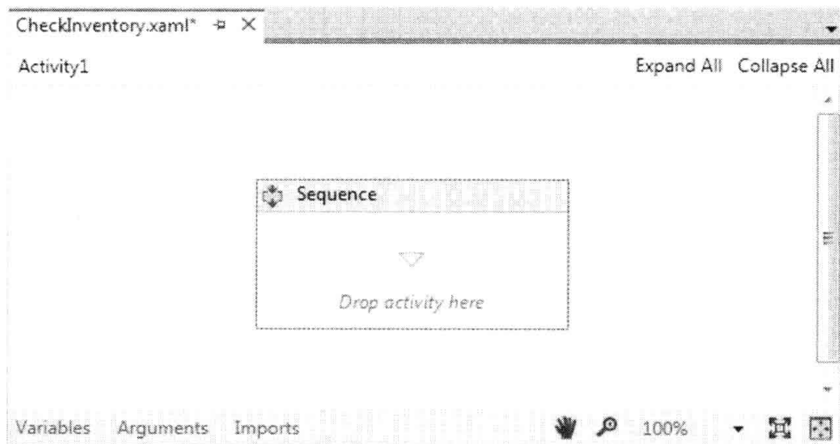


图26-21　最顶端的Sequence活动

顾名思义，Sequence活动将帮助你轻松地创建一连串逐个发生的任务。但这并不意味着子活动也必须遵循严格的线性路径。序列中可以包含流程图、其他序列、并行数据处理、if/else分支以及其他可能为正在设计的业务流程带来重要意义的活动。

26.6.2 引入程序集和命名空间

由于工作流将与AutoLot数据库进行交互，我们下一步要做的是使用Visual Studio的Add Renference对话框添加AutoLot.dll程序集的引用。我们将在本例中使用非连接层，因此我建议你引用第22章创建的程序集的最终版本（AutoLotDAL (Version 3)）。

该工作流还将使用LINQ to DataSet API查询返回的DataTable，来验证所请求的项是否有现货。因此还需要引用System.Data.DataSetExtensions.dll，因为新的Activity Library项目不会自动引入这个程序集。

添加了这些程序集之后，单击工作流设计器底部的Imports按钮。在该编辑器顶端的文本框中可以输入想要在工作流作用域中使用的.NET命名空间的名称（可以将这个区域看做是C# using 关键字的声明式版本）。

你可以在Imports编辑器最上方的文本框中输入要从引用的程序集中添加的命名空间。用该文本框引入AutoLotDisconnectedLayer和System.Data.DataSetExtensions。这样在引用所包含的类型时就不必使用完全限定名称。图26-22显示了完成之后的Imports区域。

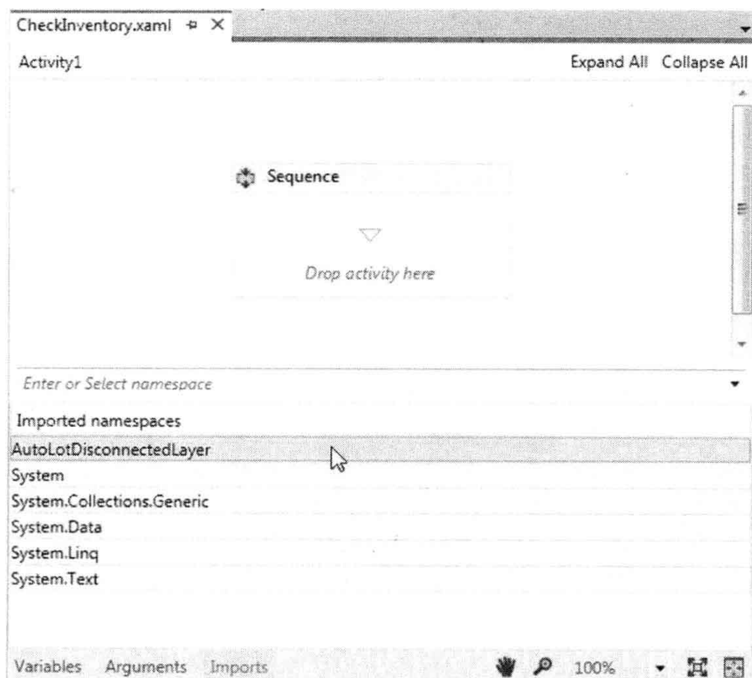


图26-22　Imports区域可以向工作流中引入.NET命名空间

26.6.3 定义工作流参数

接下来定义两个工作流输入参数，RequestedMake和RequestedColor，它们均为String类型。与上例类似，工作流的宿主程序将创建一个Dictionary对象，包含映射到这些参数的数据，因此没有必要在Arguments编辑器中指定这些项的默认值。正如你猜测的那样，工作流将使用这些传入值来执行数据库查询。

同样，你可以使用相同的Arguments编辑器定义名为FormattedResponse类型为String的输出参数。当需要工作流向宿主程序返回数据的时候，你可以创建任意数量的输出参数，这些输出参数可以在工作流完成时被宿主程序枚举。图26-23显示了当前的工作流设计器。

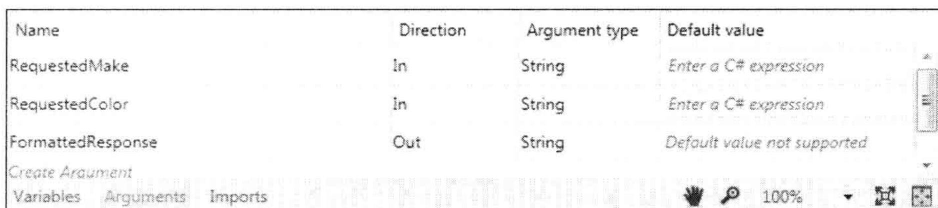


图26-23　输入和输出参数

26.6.4 定义工作流变量

现在，我们需要在工作流中声明一个成员变量，用来与AutoLotDAL.dll中的InventoryDALDisLayer类通信。我们在第22章创建的这个类可以获取Inventory中的所有数据，并保存在一个DataTable中。在设计器中选择Sequence活动，单击Variables按钮，创建一个名为AutoLotInventory的变量。在Variable Type下拉列表框中，选择Browse For Types...菜单选项，然后输入InventoryDALDisLayer（如图26-24所示）。

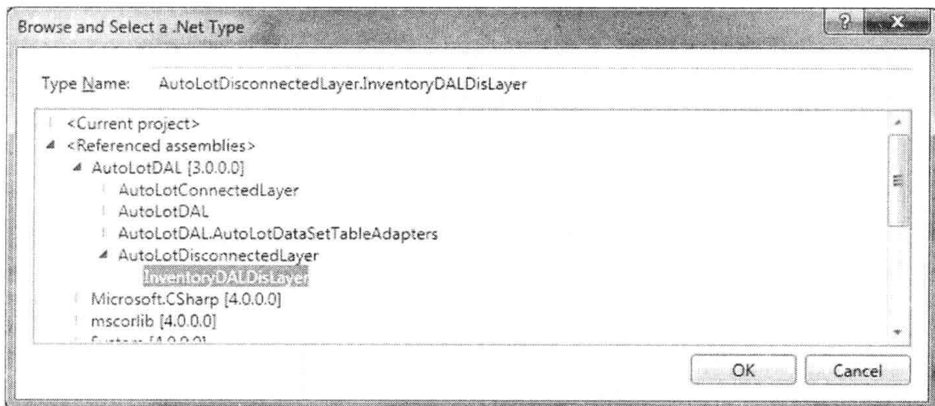


图26-24　工作流变量可以在某个作用域内以声明的方式定义变量

选中新建的变量，在Visual Studio的Properties窗口中单击Default属性旁的椭圆形按钮。这将打开一个代码编辑器，你可以根据需要调整这个编辑器的大小（在输入较长代码时非常有用）。如果在设置变量时需要输入复杂的代码，该编辑器也很好用。输入下面的（单行）代码，分配InventoryDAL-DisLayer变量：

```
new InventoryDALDisLayer(@"Data Source=(local)\SQLEXPRESS;Initial Catalog=AutoLot;Integrated
Security=Ture")
```

使用WF设计器再声明一个System.Data.DataTable类型的工作流变量Inventory，同样使用Browse For Types...菜单选项，将默认值设置为null，如图26-25所示。



图26-25　在工作流中声明一个DataTable变量

在后面的步骤中，你还要把在InventoryDALDisLayer变量上调用GetAllInventory()的结果赋给Inventory变量。

26.6.5　使用Assign活动

Assign活动可以设置变量的值，该值可以为任何有效的代码语句的输出结果。将Assign活动（位于Toolbox的Primitives区域内）拖曳到Sequence活动中（如图26-26所示）。

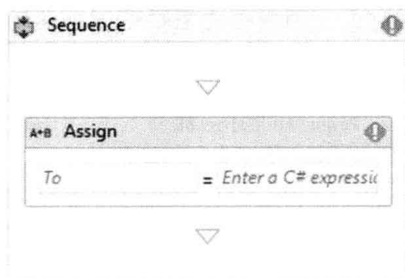


图26-26 Assign活动

在左侧的编辑框中指定Inventory变量。在右侧的编辑框中输入如下代码：

```
AutoLotInventory.GetAllInventory()
```

当工作流执行到Assign活动时，将得到包含所有Inventory表中记录的DataTable。但我们所需要的是确定与宿主程序发送的RequestedMake和RequestedColor值相符的项是否存在于库存中。这需要使用LINQ to DataSet API和一个If工作流活动。

26.6.6 使用If和Switch活动

将一个If活动拖曳到Sequence节点，放置在Assign活动下面（如图26-27所示）。

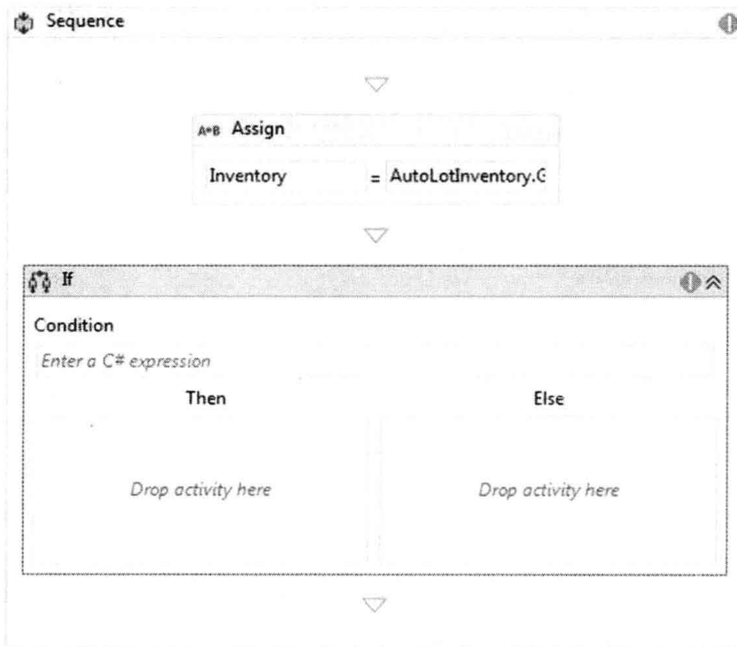


图26-27 If活动

由于If活动可以进行运行时决策，因此首先需要配置If活动用于验证的Boolean表达式。在Condition编辑器中，输入如下的验证LINQ to DataSet查询：

```
(from car in Inventory.AsEnumerable()
   where (string)car["Color"] == RequestedColor &&
   (string)car["Make"] == RequestedMake select car).Any()
```

该LINQ查询使用工作流宿主提供的RequestedColor和RequestedMake参数获取DataTable中所有符合所选品牌和颜色的记录。然后调用Any()扩展方法，根据查询是否有结果而返回true或false。

下一步的任务是配置一些当指定条件为true或false时将执行的任务。记住我们的最终目标是，如果存在所请求的汽车，则向用户返回一个格式化的消息。不过，为了增加一些趣味性，我们返回的消息根据调用者所请求的汽车品牌（如BMW、Yugo等）的不同而有所区别。

将一个Switch<T>活动（位于Toolbox的Flow Control区域）拖曳到If活动的Then区域。这时，Visual Studio将立即显示一个对话框，询问泛型类型参数的类型。这里我们指定为String。这时，使用工作流设计器来设置Switch活动的Expression字段，输入RequestedMake（如图26-28所示）。

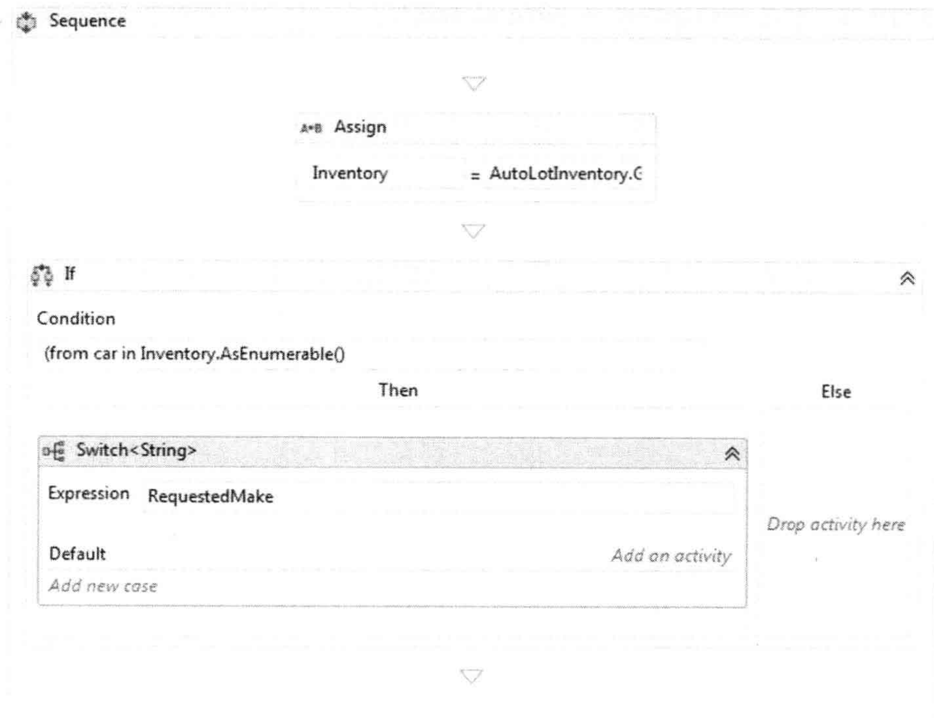


图26-28　Switch活动

你将看到Switch活动的默认选项，但必须展开才能添加子活动（单击“Add an Activity”）。在Default编辑域中添加一个单独的Assign活动，然后用如下的代码语句设置FormattedResponse参数：

```
String.Format("Yes, we have a {0} {1} you can purchase",
              RequestedColor, RequestedMake)
```

这时，Switch编辑器将如图26-29所示。

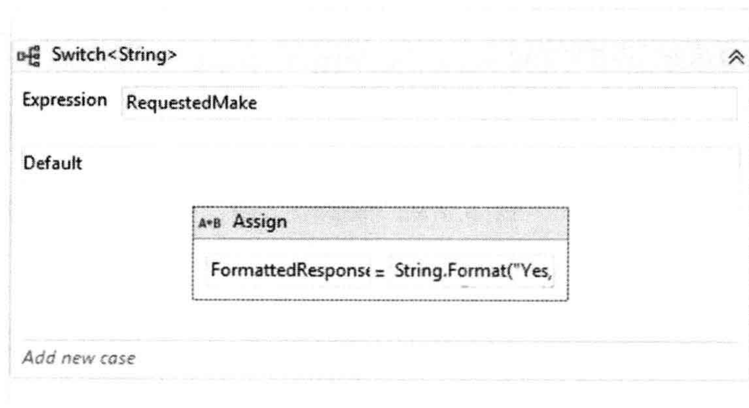


图26-29　定义Switch活动的默认任务

现在，单击“Add New Case”链接，在第一个事例（case）处输入BMW（不加双引号），然后再添加一个Yugo事例（同样不加双引号）。在这些事例区域中，分别拖曳一个Assign活动，并且都为Formatted-Response变量赋值。对于BMW事例，赋值语句如下：

```
String.Format("Yes sir! We can send you {0} {1} as soon as {2}!",
            RequestedColor, RequestedMake, DateTime.Now)
```

对于Yugo事例，使用如下的表达式：

```
String.Format("Please, we will pay you to get this {0} off our lot!",
            RequestedMake)
```

现在Switch活动的外观大致如图26-30所示。

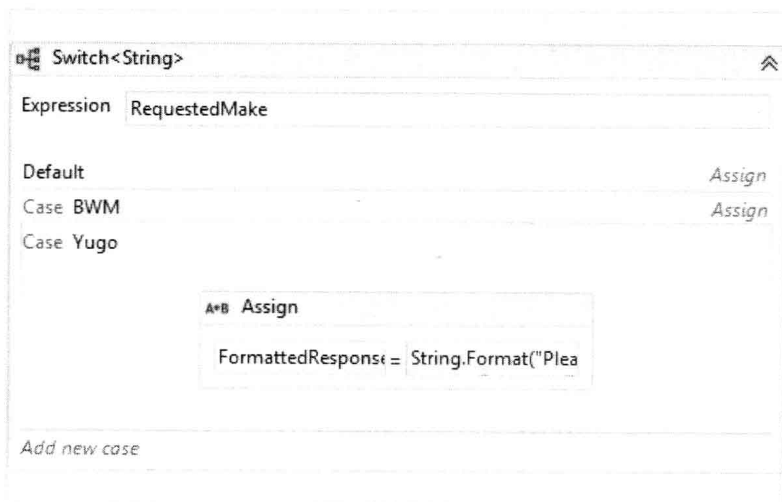


图26-30　最终的Switch活动

26.6.7　构建自定义代码活动

与工作流设计器体验一样富于表现力的是，你可以在XAML文件中内嵌复杂的代码语句（和LINQ查询），在专门的类中编写代码，这样的需求是很常见的。使用WF API有很多种方法可以实现这个目标，但最直接的方式是创建一个类扩展`CodeActivity`，或者如果活动需要返回值的话，创建一个`CodeActivity<T>`（T为返回值类型）。

在这里，我们将创建一个简单的自定义活动，将数据转储到文本文件中，并通知销售人员库存系统中没有当前请求的汽车。首先，单击Project | Add New Item菜单选项，插入一个Code Activity，取名为CreateSalesMemoActivity.cs（如图26-31所示）。

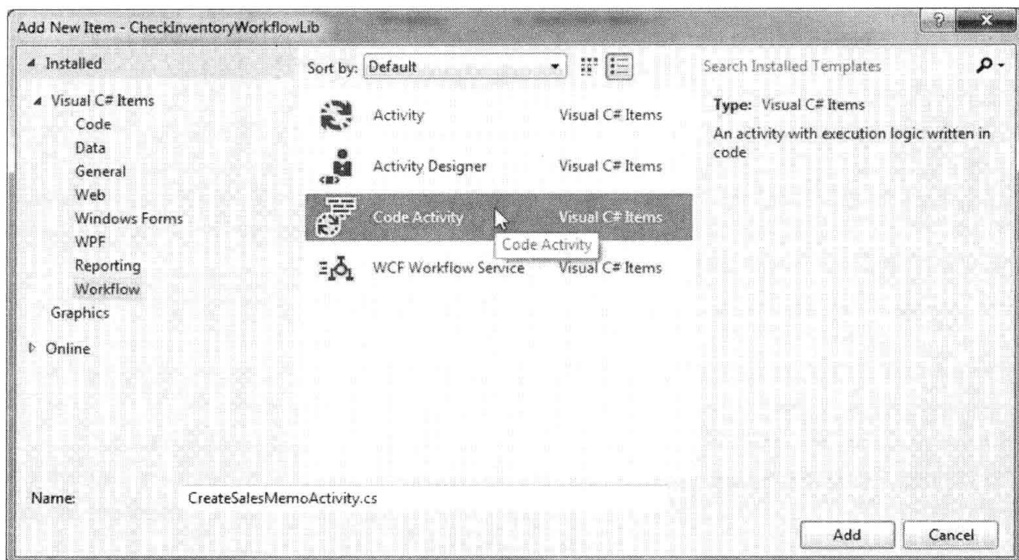


图26-31　插入新的Code Activity

如果自定义活动需要输入内容，可以用`InArgument<T>`类型的属性来表示。`InArgument<T>`类是WF API特有的实体，可以将工作流提供的数据传入自定义活动类的内部。我们的活动需要两个这样的属性，分别表示没有现货的汽车的品牌和颜色。

同样，自定义代码活动需要重写虚方法`Execute()`，当执行到该活动时，WF运行时会调用该方法。该方法将使用`InArgument<>`属性。为了得到实际的值，需要间接使用`CodeActivityContext`的`GetValue()`方法。

以下是自定义活动的代码，它将生成一个*.txt文件，用来向销售团队描述情况：

```
public sealed class CreateSalesMemoActivity : CodeActivity
{
  // 自定义活动的两个属性
  public InArgument<string> Make { get; set; }
  public InArgument<string> Color { get; set; }
```

```
    // 如果活动有返回值，则需要继承CodeActivity<TResult>，并在Execute方法中返回该值
    protected override void Execute(CodeActivityContext context)
    {
      // 将消息转存到本地文本文件中
      StringBuilder salesMessage = new StringBuilder();
      salesMessage.AppendLine("***** Attention sales team! *****");
      salesMessage.AppendLine("Please order the following ASAP!");
      salesMessage.AppendFormat("1 {0} {1}\n",
        context.GetValue(Color), context.GetValue(Make));
      salesMessage.AppendLine("*********************************");

      System.IO.File.WriteAllText("SalesMemo.txt", salesMessage.ToString());
    }
  }
```

编译工作流程序集。确保工作流设计器在Visual Studio IDE中处于激活状态。检查Toolbox的上方，将看到我们的自定义活动及其说明（如图26-32所示）。

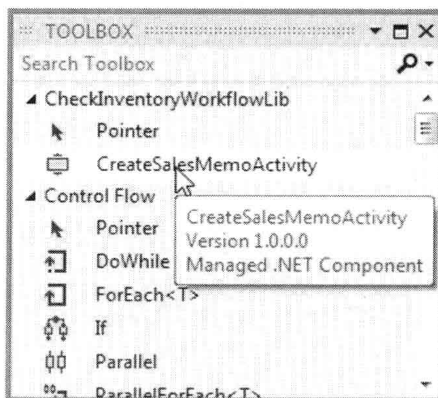


图26-32 自定义代码活动出现在Visual Studio的Toolbox中

首先，将一个新的Sequence活动拖曳到If活动的Else分支。然后，将自定义活动拖曳到Sequence。这时，我们可以在Properties窗口中设置暴露出来的属性。用RequestedMake和RequestedColor变量设置活动的Make和Color属性，如图26-33所示。

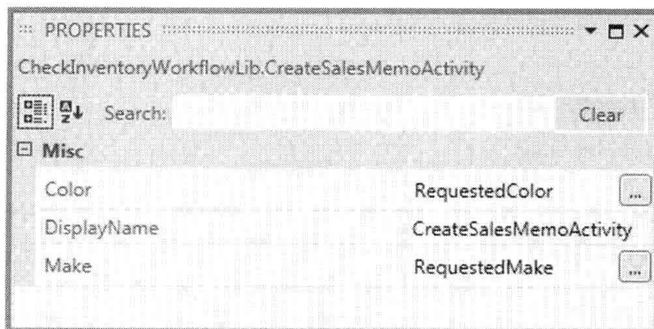


图26-33 设置自定义代码活动的属性

最后，拖曳一个Assign活动到Else分支的Sequence活动中，并设置FormatedResponse的值为“Sorry, out of stock”。图26-34显示了最终的If活动。

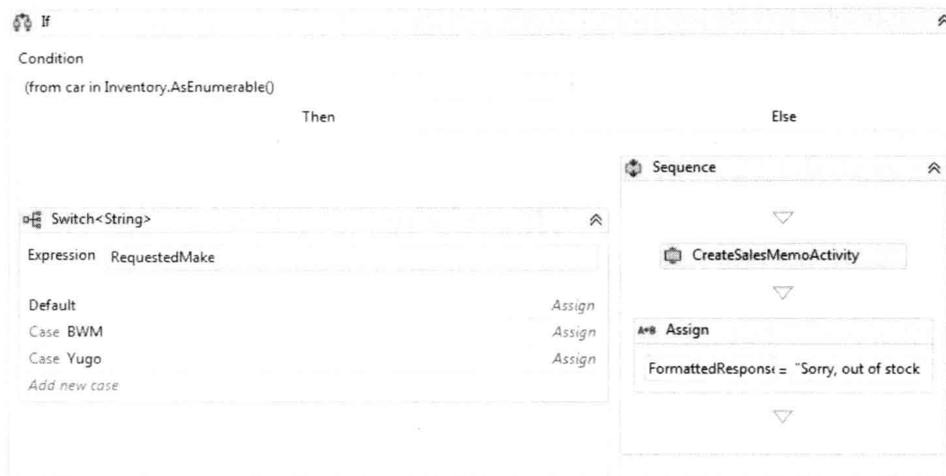


图26-34　完整的If活动

编译项目，接下来是本章的最后一部分，我们将构建一个使用该工作流的客户端宿主程序。

源代码　CheckInventoryWorkflowLib项目的源代码位于Chapter 26子目录下。

26.7　使用工作流库

任何类型的应用程序都可以使用工作流库。但这里我们简便起见，创建一个工作台应用程序，命名为WorkflowLibraryClient。建好之后，需要添加的引用除了CheckInventoryWorkflowLib.dll和AutoLotDAL.dll，还包括WF主库System.Activities.dll。将这些库添加到引用。

然后，用下面的逻辑更新Program.cs文件：

```
using System;
...
using CheckInventoryWorkflowLib;

namespace WorkflowLibraryClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("**** Inventory Look up ****");

      // 获取用户偏好
      Console.Write("Enter Color: ");
      string color = Console.ReadLine();
```

```
            Console.Write("Enter Make: ");
            string make = Console.ReadLine();

            // 包装工作流要用的数据
            Dictionary<string, object> wfArgs = new Dictionary<string, object>()
            {
              {"RequestedColor", color},
              {"RequestedMake", make}
            };

            try
            {
              // 向工作流发送数据
              WorkflowInvoker.Invoke(new CheckInventory(), wfArgs);
            }
            catch (Exception ex)
            {
              Console.WriteLine(ex.Message);
            }
          }
        }
      }
```

与其他示例一样，我们使用WorkflowInvoker以同步的方式调用工作流。尽管这一切看上去都不错，但我们如何获取工作流的返回值呢？记住，当工作流终止的时候，我们应该得到格式化的响应。

获取工作流的输出参数

WorkflowInvoker.Invoke()方法将返回一个实现了IDictionary<string, object>接口的对象。由于工作流可以返回任意数量的输出参数，因此你需要用一个字符串值来指定每个输出参数的名称，来作为类型索引器。用如下代码更新try/catch逻辑：

```
try
{
  // 向工作流发送数据
  IDictionary<string, object> outputArgs =
    WorkflowInvoker.Invoke(new CheckInventory(), wfArgs);

  // 打印输出的消息
  Console.WriteLine(outputArgs["FormattedResponse"]);
}
catch (Exception ex)
{
  Console.WriteLine(ex.Message);
}
```

现在，返回程序，输入一个已经存在于AutoLot数据库Inventory表中的汽车的品牌和颜色，将看到如下的输出结果：

```
**** Inventory Look up ****
Enter Color: Black
Enter Make: BMW
Yes sir! We can send you Black BMW as soon as 2/17/2012 9:23:01 PM!
Press any key to continue . . .
```

而如果输入不存在的项，将只能看到如下的输出结果：

```
**** Inventory Look up ****
Enter Color: Pea Soup Green
Enter Make: Viper
Sorry, out of stock
Press any key to continue . . .
```

你还将在客户端应用程序的\bin\Debug文件夹下找到*.txt文件。打开该文件，将看到如下所示的“销售备忘”：

```
***** Attention sales team! *****
Please order the following ASAP!
1 Pea Soup Green Viper
*********************************
```

这就是我们对于WF API的简要介绍。由于本章只接触了.NET平台中这个特殊领域的一些主要方面，我真诚地希望如果你感兴趣的话能够有信心深入研究这个话题。

源代码　WorkflowLibraryClient项目的源代码位于Chapter 26子目录下。

26.8　小结

从本质上讲，WF允许你在应用程序内直接对应用程序的内部工作流进行建模。但是，除了简单地对工作流建模，WF还提供了完整的运行时引擎和多项服务，以便用来完成该API的所有功能（持久化、跟踪服务等）。虽然本章没有直接介绍这些服务，但要知道的是一个产品级别的WF应用程序肯定会使用这些工具。

在本章的介绍中，你学习到了两个顶级的活动Flowchart和Sequence。虽然这两个类型以不同的方式控制逻辑流，但他们可以包含相同种类的子活动，并且以相同的方式被宿主程序执行（通过WorkflowInvoker或WorkflowApplication）。你还学习了如何使用泛型Dictionary对象向工作流传入宿主参数，以及如何使用泛型IDictionary兼容的对象从工作流获取输出参数。

Part 7 第七部分

WPF

本部分内容

第27章 WPF和XAML

在.NET 1.0中，建立图形化桌面应用程序可以使用两个API：Windows Forms和GDI+，它们主要位于System.Windows.Forms.dll和System.Drawing.dll程序集内。尽管Windows Forms/GDI+对于构建传统的桌面GUI来说是非常优秀的API，但微软在.NET 3.0中发布了另一个GUI桌面API——WPF。

作为学习WPF的第一章，本章首先研究这个全新的GUI框架背后的动机，这将有助于你理解Windows Forms/GDI+和WPF编程模型的区别。接下来，我们将介绍该API所支持的不同类型的WPF应用程序，并理解一些重要类的作用，包括`Application`、`Window`、`ContentControl`、`Control`、`UIElement`和`FrameworkElement`。在此期间，你还将学习只使用C#代码捕获键盘和鼠标活动，定义应用程序范围的数据，以及其他常见WPF任务。

这一章余下部分会介绍一种全新的基于XML的语法，即XAML（Extensible Application Markup Language，可扩展应用程序标记语言，读作“zammel”）。在这里，我们会介绍XAML的语法和语义（包括附加属性语法、类型转换的作用、标记扩展），并理解如何在运行时生成、加载和解析XAML。同样，你还将学习如何将XAML数据整合到WPF的C#代码库（以及这么做的好处）。

本章最后介绍了Visual Studio中集成的WPF设计器。在此，我们将构建一个自定义的XAML编辑器/解析器，可以演示如何在运行时操纵XAML来动态地构建用户界面。

27.1 WPF背后的动机

近年来，微软已经开发出了众多的GUI开发工具包（原始的C/C++/Windows API开发、VB6、MFC，等等）用以创建桌面可执行程序。这些GUI API各自都提供了用以代表GUI应用程序基本要素的代码库，这些基本要素包括主窗体、对话框、控件、菜单系统以及其他一些必需的要素。随着.NET平台的发布，Windows Forms API开发模型凭借其简单而又强大的对象模型，快速成为UI开发的首选。

虽然许多功能完整的桌面应用程序已经通过使用Windows Forms成功地开发出来，但事实上现在的桌面开发编程模型是非常不对称的。简单地说，System.Windows.Forms.dll和System.Drawing.dll没有对创建完全成熟的桌面应用程序所需的许多技术提供直接的支持。为了说明这一点，考虑一下在WPF发布之前进行GUI桌面开发时的混乱局面（如表27-1所示）。

正如你所看到的，Windows Forms开发人员必须引入来自多种不同API和对象模型的类型。尽管使用这些不同种类的API可能在语法上看起来比较相似（毕竟都是C#代码），但你也应该承认每种技术都需要截然不同的思维方式。比如，使用DirectX创建三维呈现动画时所需要的技巧，与将数据绑定到网

格时所需要的技巧完全不一样。肯定地讲，要一个Windows Forms编程人员掌握所有这些迥异的API是非常困难的。

表27-1 WPF之前为所需功能提供的解决方案

所要求的功能	技 术
构建带控件的表单	Windows Forms
2D图形支持	GDI+（System.Drawing.dll）
3D图形支持	DirectX API
对流视频的支持	Windows Media Player API
对流文档的支持	编程操作PDF文件

27.1.1 统一多种不同的API

WPF（在.NET 3.0中引入的）正是本着要将这些之前不相关的编程任务融合为一个统一对象模型的目的而开发出来的。这样，如果你需要创建一段三维动画，就没有必要使用DirectX API进行手工编程，因为这个功能已经被直接内嵌在WPF中了。要了解情况变得何等简化，如表27-2所示，它简要说明了.NET 3.0桌面开发模型。

表27-2 NET 3.0对所要求的功能的解决方案

所要求的功能	技 术
构建带控件的表单	WPF
2D图形支持	WPF
3D图形支持	WPF
对流视频的支持	WPF
对流文档的支持	WPF

一个显而易见的好处是，现在.NET开发者拥有一个单一、对称的API，可以满足所有常见的GUI桌面程序的需要。在学习完WPF主要程序集的功能和XAML的语法之后，你会惊诧于创建一个复杂的UI是如此快速。

27.1.2 通过XAML将关注点分离

也许WPF最引人注目的好处之一就是，它提供了一种方法，能将GUI应用程序的界面外观与驱动它们的编程逻辑清晰地相互分离。使用XAML，可以利用标记（markup）来定义一个应用程序的UI。这种标记（理想状态下由Microsoft Visual Studio或Microsoft Expression Blend工具来创建）随后可以连接到一个相关的C#代码文件，由它提供相应的程序功能。

WPF另外一个引人注目的特点就是这种“桌面标记”所提供的灵活性。在XAML标记中，你不仅可以定义简单的UI元素（按钮、网格、列表框等），而且还可以定义交互的二维和三维图像、动画、数据绑定逻辑以及多媒体功能（如视频回放）。

说明 XAML不仅仅局限于WPF应用程序。任何应用程序都可以使用XAML来描述.NET对象树，即使和可视化用户界面没有什么关系。例如，可以使用基于XAML的语法为WF API定义业务流程和自定义活动。同样，其他.NET GUI框架，如Silverlight、Windows Phone 7和Windows 8应用程序，都使用了XAML。

XAML还大大简化了自定义控件如何呈现其可视外观。比如，定义一个以一家公司商标为动画背景的圆形按钮控件，只需短短几行标记语言。如第31章中所示，WPF控件还可以通过样式和模板来加以修饰，这样就使你能够独立于核心程序来处理代码，花最少精力就可改变一个应用程序整体的界面外观。和Windows Forms开发不同，从头构建自定义WPF控件的唯一可能的原因就是我们需要改变控件的行为（如增加自定义方法、属性或事件、重写既有控件的虚拟成员等）。如果你只需要改变控件的观感（又如圆形按钮），完全可以通过标记来实现。

27.1.3　提供优化的呈现模型

Windows Forms、MFC、VB6这样的GUI工具在执行所有的图形呈现请求（包括呈现按钮、下拉框这样的UI元素）时，都使用一个低级别的、基于C的API（GDI），它作为Windows操作系统的一部分已经很多年了。GDI为典型的商业应用或简单的图形程序提供了足够的性能，但如果UI应用程序需要更高性能的图形，就需要DirectX了。

WPF编程模型则完全不同，它在呈现图形数据时不使用GDI。所有呈现操作（如2D图形、3D图形、动画、控件呈现等）都使用DirectX API。最显而易见的好处是，WPF应用程序将自动利用软硬件优化的优势。同样，WPF应用程序还能使用丰富的图形服务（模糊效果、抗锯齿、透明度等），并且避免了直接使用DirectX API进行编程所带来的复杂性。

说明 尽管WPF将所有的呈现请求都推入DirectX层，但这并不意味着WPF应用程序能达到直接使用非托管C++和DirectX时的执行速度。如果要构建最快速的桌面应用（如3D视频游戏），非托管C++和DirectX仍然是最佳方案。

27.1.4　简化复杂的UI编程

总结一下，WPF（Windows Presentation Foundation）是一个新的API，它可以构建将各种桌面API整合到一个对象模型中的桌面应用程序，并通过XAML提供彻底的关注点分离。除了这些要点，WPF还能以十分简单的方式在程序中集成服务，而这在之前是相当复杂的。下面是这些核心WPF特性的简介。

- 众多对放置操作和内容重定位提供完全支持的布局管理器（远远多于Windows Forms）。
- 使用增强型的数据绑定引擎，支持多种方式的内容到UI元素的绑定。
- 一个内置的样式引擎，允许你为WPF程序定义“主题”。
- 使用矢量图形，允许图像自动调整大小来适应承载应用程序的屏幕的大小和分辨率。
- 支持2D和3D图形、动画、视频以及音频回放。
- 丰富的印刷API，如支持XML文件规范（XPS）文档、固定文档（WYSIWYG）、流文档以及文档注释（如粘滞便笺API）。

- ❑ 支持与遗留GUI模型（如Windows Forms、ActiveX以及Win32 HWND）的互操作。例如，可以在WPF应用程序中应用自定义的Windows Forms控件，反之亦然。

现在我们已经对WPF有了初步的认识，下面我们来看看可以使用该API创建的不同类型的应用程序。放心，这些特性将会在后面的章节里详细介绍。

27.2 各种形式的 WPF 应用程序

WPF API可以用于构建各种GUI相关的应用程序，它们在导航结构和部署模型方面都不同。这部分内容会大致介绍每一个选项。

27.2.1 传统的桌面应用程序

第一个（最熟悉）选项是运行于本地计算机的传统的可执行程序集。例如，我们可以使用WPF来构建文本编辑器、绘图程序或者诸如数字播放器、图片查看器等多媒体程序。和其他桌面应用程序相似，这些*.exe文件可以使用传统的方式进行安装（安装程序、Windows安装包等）或通过ClickOnce技术允许桌面应用程序通过远程Web服务器分布和安装。

从编程角度说，这个类型的WPF应用程序（至少）会使用Window和Application类类型，以及对话框、工具条、状态栏、菜单系统和其他UI元素。

现在，你肯定可以使用WPF构建这种基本的、没有任何特效的商业应用，但只有加入这些特性时，WPF才会真正闪光。图27-1显示了一个WPF示例桌面应用，可以查看医疗系统中患者记录。



图27-1　使用了一些WPF API的WPF桌面应用程序

遗憾的是，截屏无法显示该程序的全部特性。比如，如果你能看到运行中的该程序，注意主窗口的右上方显示的是患者窦性心率的实时图像。单击右下方的Patient Details按钮，将启动一些动画，使UI翻转、旋转、变换成如图27-2所示的外观。



图27-2　WPF中的变换和动画是十分简单的

我们能用WPF创建一个相同的应用程序吗？当然可以。但是代码很多也很复杂。

说明　该示例应用以及很多其他应用都可以在WPF的官方网站http://windowsclient.net上下载。你可以在这个网站上找到很多WPF（和Windows Forms）白皮书、示例项目、技术演练和讨论。

27.2.2　基于导航的WPF应用程序

WPF应用程序可以选择使用基于导航的结构，它是基于Web浏览器应用程序的传统桌面应用程序。使用这个模型，我们可以构建桌面*.exe来提供“向前”和“向后”按钮，以允许终端用户在各种页面之间切换。

这种类型的应用程序维护了每一个页面的列表，并且提供了在它们之间导航、在页面之间传递数据（和基于Web的应用程序变量相似）以及维护历史列表的必要基础结构。Windows资源管理器（如图27-3所示）是一个实际的例子，它就使用了这样一种功能。注意窗体左上方的导航按钮（以及历史列表）。

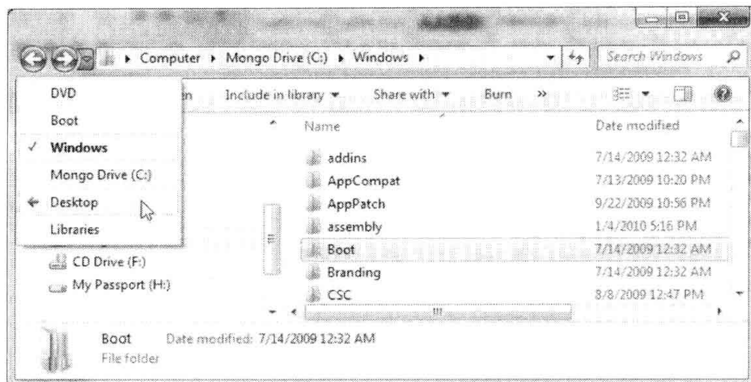


图27-3 基于导航的桌面程序

尽管WPF桌面应用程序可以使用和Web相似的导航结构，但这只是UI设计问题。应用程序本身只是运行于桌面计算机的可执行程序，并且除了看上去有点相似之外，和Web应用程序没什么相关性。从编程角度来说，这类WPF应用程序使用诸如`Application`、`Page`、`NavagationWindow`和`Frame`的类进行表示。

27.2.3 XBAP应用程序

WPF还允许我们构建可以在Web浏览器中承载的应用程序。这种形式的WPF应用程序叫做XAML浏览器应用程序或XBAP。在这个模型下，终端用户导航到某个URL，XBAP应用程序（从本质上说，它是`Page`对象的集合）从这个点透明下载并且安装在本地机器上。然而和可执行应用程序传统的ClickOnce不同的是，XBAP程序直接在浏览器中承载并且采用了浏览器的原生导航系统。图27-4演示了运行中的XBAP程序（ExpenseIt WPF示例程序，可以在http://windowsclient.net找到）。

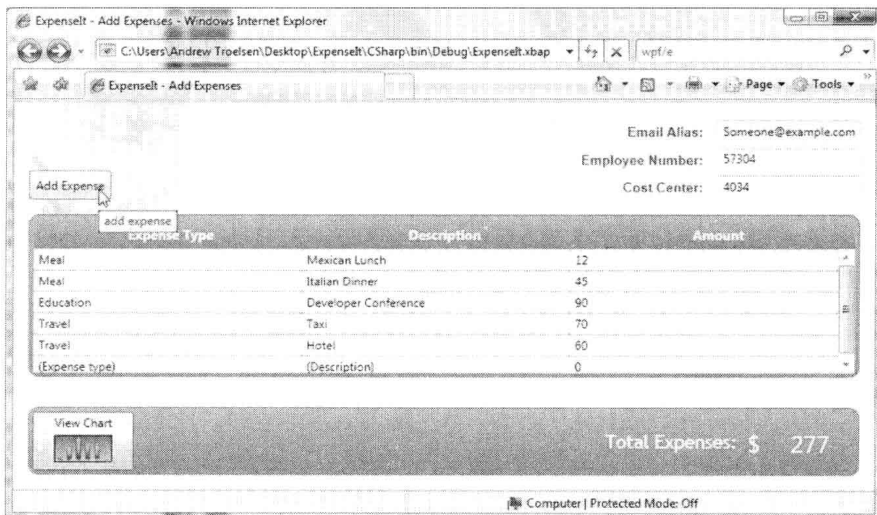


图27-4 XBAP程序下载到本地机器并且在Web浏览器中进行承载

XBAP的一个好处是，与用HTML和JavaScript构建的典型Web页面相比，它创建的UI更复杂，也更富于表现力（当然，HTML5改善了这一局面）。一个XBAP Page对象可以像在桌面WPF应用程序中那样使用WPF服务，包括动画、二维和三维图形、主题等。实际上，Web浏览器只是WPF Page对象的容器，而不是在显示ASP.NET页面。

不过，这些Page对象部署在远程Web服务器上，XBAP可以轻松地对它们进行版本化和更新，而不需要在用户桌面上重新部署可执行文件。与传统Web程序类似，你可以简单地更新Web服务器上的Page对象，用户访问URL时将得到“最新最好”的页面。

这种形式的WPF可能有一个缺点，XBAP必须承载于微软IE或FireFox浏览器中。如果我们在公司局域网中部署这样的XBAP，浏览器兼容性可能不是问题，因为系统管理员可以决定用户的机器上需要安装哪个浏览器。然而，如果你希望外部世界能使用XBAP的话，就不可能确保每一个终端用户都使用IE或FireFox，因此一些外部用户可能就不能查看我们的WPF XBAP应用程序。

另一个要注意的问题是，浏览XBAP的计算机必须安装.NET Framework，因为Page对象需要像本地运行的应用程序那样使用.NET程序集。因此，XBAP仅限于Windows操作系统，不能在Mac OS X或Linux上浏览。

27.2.4 WPF/Silverlight关系

WPF和XAML还提供了一个叫做Silverlight的基础结构，用于WPF技术的跨平台和跨浏览器。你可以将Silverlight看成是Adobe Flash的竞争者，它使用的是C#和XAML，而不是新的工具和语言。Silverlight是WPF功能的子集，可以为大型HTML Web页面构建高交互性的插件。然而实际上，Silverlight是.NET平台一个完全独立的部分，它包含“迷你”的CLR和.NET基础类库的“迷你”版本。

与XBAP不同，用户计算机不必完整安装.NET Framework。只需要安装Silverlight运行时，浏览器就可以自动加载Silverlight运行时并显示Silverlight应用程序。更妙的是，Silverlight插件不局限于Windows操作系统。微软也为Mac OS X创建了Silverlight运行时。

使用Silverlight，我们可以构建功能非常丰富的（交互式）Web应用程序。例如，和WPF相似，Silverlight有基于向量的图形系统、动画支持以及多媒体支持。此外，我们可以把.NET基础类库的子集集成到我们的应用程序中。这个子集包括了LINQ API、泛型集合、WCF支持以及mscorlib.dll子集（文件I/O、XML操作等）。

说明　本书不会介绍Silverlight，但是大部分WPF知识会直接映射到SilverLight Web插件的构建。如果你希望更了解这个API的话，可以访问www.silverlight.net。

27.3 WPF程序集

不管我们希望构建何种类型的WPF应用程序，WPF最终只是.NET程序集中包含的一些类型的集合。表27-3描述了用于构建WPF应用程序的主要程序集，每一个都会在新建项目的时候被引用。正如你期望的那样，Visual Studio WPF项目会自动引用必要的程序集）。

表27-3 WPF核心程序集

程序集	作用
PresentationCore.dll	这个程序集定义了许多构成WPF GUI层基础的命名空间。例如，这个程序集包含了WPF Ink API（用于Pocket PC和Tablet PC笔针输入的编程）的支持、动画基元以及几个图形渲染类型
PresentationFramework.dll	这个程序集包含大量WPF控件、Application和Window类、对交互的二维几何图形的支持以及大量用于数据绑定的类型
System.Xaml.dll	该程序集提供的命名空间允许在运行时对XAML文档进行编程。总体上来说，只有在编写WPF支持工具或需要在运行时完全控制XAML时，才会用到这个库
WindowsBase.dll	这个程序集定义了构成WPF API基础结构的核心类型，其中包括表示WPF线程类型、安全类型、各种类型转换器以及对依赖属性和路由事件的支持（见第31章）

总之，这4个程序集定义了许多新的命名空间以及数以百计的新的.NET类、接口、结构体、枚举以及委托。你可以查看.NET Framework 4.5 SDK文档来了解全部细节，表27-4列出了你需要留意的部分核心命名空间的作用。

表27-4 WPF核心命名空间

命名空间	作用
System.Windows	这是WPF的根命名空间。在这里你将找到所有WPF桌面项目所需要的核心类（如Application类和Window类）
System.Windows.Controls	包括用于构建菜单系统、工具提示以及众多布局管理器的多种类型
System.Windows.Data	包含用于WPF数据绑定引擎的类型，并且支持数据绑定模板
System.Windows.Documents	包含用于文档API的类型，可以通过XML Paper Specification（XPS）协议，在WPF应用程序中集成PDF样式功能
System.Windows.Ink	支持Ink API，可用于捕获手写笔或鼠标的输入，响应输入笔势等。主要用于平板电脑编程，但所有WPF应用都可以使用该API
System.Windows.Markup	这个命名空间定义了一些用来解析和编程处理XAML标记（以及等价的二进制格式，BAML）的类型
System.Windows.Media	这是多个以媒体为主的命名空间的根空间。在这些命名空间中，你将找到那些用于动画、三维显示、文本显示以及其他多媒体用途的类型
System.Windows.Navigation	这个命名空间提供了解释XAML浏览器程序（XBAP）和需要导航页面模型的标准桌面应用程序所用的导航逻辑的多种类型
System.Windows.Shapes	这个命名空间定义了一些类，允许呈现自动响应鼠标输入的交互式二维图形

在开始介绍WPF编程模型之前，我们先来研究一下在所有传统桌面开发中都很常见的System.Windows命名空间中的两个成员，Application类和Window类。

说明 如果你曾经使用Windows Forms API创建过桌面UI，要注意System.Windows.Forms.*和System.Drawing.*程序集与WPF无关。这些库表示原来的.NET GUI工具——Windows Forms/GDI+。

27

27.3.1 Application类的作用

System.Windows.Application类代表了一个运行中的WPF应用程序的全局实例。这个类提供了一个Run()方法（用以启动这个应用程序）、一系列可处理的事件（如Startup和Exit，用于在程序生命期内与其进行交互）以及一些专为XAML浏览器程序设置的成员（如为页面间的用户导航而触发的事件）。表27-5列出了其一部分主要成员。

表27-5 Application类型的关键属性

属 性	作 用
Current	这个静态属性让你能够在代码中的任何地方访问正在运行的Application对象。当窗口或对话框需要访问创建它的Application对象时，特别是访问应用程序范围内的变量和函数时，Current很有用
MainWindow	这个属性允许你通过编程来获取或设置应用程序的主窗口
Properties	这个属性使你可以建立和获取整个WPF应用程序中可以访问的数据（如窗口、对话框等）
StartupUri	这个属性可以获取或设置一个URI，指定在应用程序启动时自动打开的窗口或页面
Windows	这个属性返回一个WindowCollection类型，通过它可以访问由创建当前Application对象的线程所创建的每个窗口。在迭代应用程序的每个打开窗口和改变其状态（如最小化所有窗口）时，这个属性会起作用

1. 构造Application类

任何WPF应用程序都需要定义扩展自Application的类。在该类中，你将定义程序的入口点（Main()方法），创建该子类的实例并处理Startup和Exit事件。稍后，我们将构建完整的示例项目，在此我们先来看一个快速示例：

```
// 为WPF程序定义全局应用程序对象
class MyApp : Application
{
  [STAThread]
  static void Main(string[] args)
  {
    // 创建应用程序对象
    MyApp app = new MyApp();

    // 注册Startup/Exit事件
    app.Startup += (s, e) => { /* Start up the app */ };
    app.Exit += (s, e) => { /* Exit the app */ };
  }
}
```

在Startup事件处理程序中，通常会处理一些传入的命令行参数，并启动程序的主窗口。Exit事件处理程序可以为程序编写任何必要的关闭逻辑（如保存用户首选项，写Windows注册表）。

2. 枚举Windows集合

Application公开的另一个有趣的属性是Windows，可以用来访问当前WPF应用程序加载到内存中的窗口集合。在新建Window对象时，会自动将它们添加到Application.Windows集合。下面的示例方法将应用程序中的所有窗口最小化（可能是响应用户触发的键盘手势菜单选项）：

```
static void MinimizeAllWindows()
{
  foreach (Window wnd in Application.Current.Windows)
  {
    wnd.WindowState = WindowState.Minimized;
  }
}
```

你将在接下来的示例中创建一个完整的派生于Application的类型。在那之前，让我们先来查看一下Window类型的核心功能和主要的WPF基类。

27.3.2 Window 类的作用

System.Windows.Window类（位于PresentationFramework.dll程序集）表示继承自Application的类所拥有的一个窗口，包括由主窗口显示的所有对话框。可能你已经想到了，Window类有一系列的父类，每个父类都有许多功能。请看图27-5，其中显示了通过Visual Studio对象浏览器看到的System.Windows.Window类型的继承关系链（以及实现的接口）。

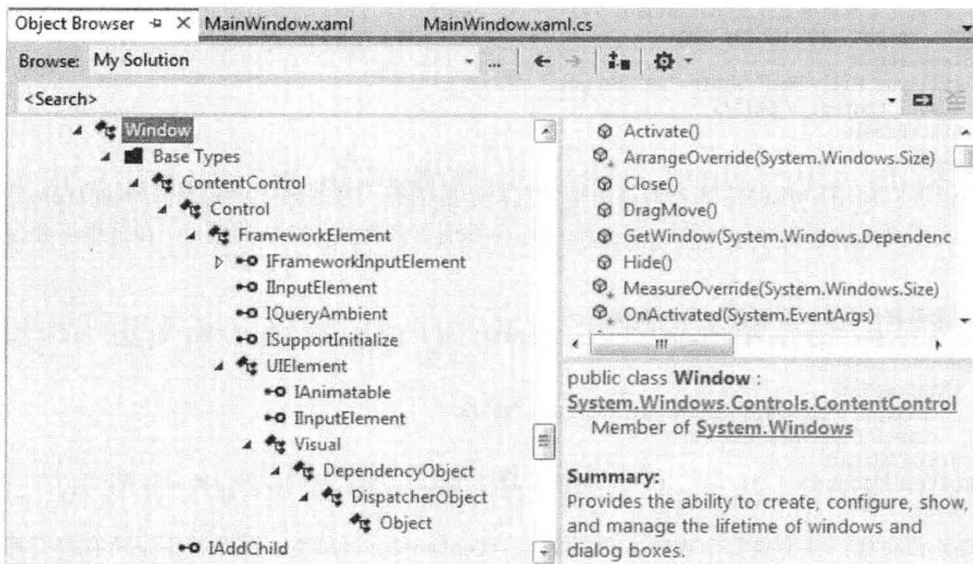


图27-5 Window类的层次结构

在学习本章的过程中，你会逐渐理解这些基类提供的各种功能。不过为了暂时吊一下大家的胃口，后面的几节将分解每个基类的作用（要了解完整的细节，请参见.NET Framework 4.5 SDK文档）。

1. System.Windows.Controls.ContentControl的作用

Window类的直接父类是ContentControl类，而该类是所有WPF类中最迷人的。这个基类为它派生的类型提供了承载内容（简单地说，就是一组通过Content属性放在控件表面的可视数据）的能力。WPF内容模型大大简化了自定义内容控件的基本外观。

例如，对于典型的“按钮”控件，其内容往往是简单的字符串（OK、Cancel、Abort等）。如果使

用XAML来描述WPF控件，并希望设置Content属性的值为简单字符串，可以像下面这样在元素的开放式定义中设置Content属性（不必担心确切的标记）：

```
<!--在开放式元素中设置Content的值-->
<Button Height="80" Width="100" Content="OK"/>
```

说明　也可以在C#代码中设置Content属性，这样就可以在运行时改变控件的内容。

但是，内容几乎可以是任何东西。例如，假设你希望“按钮”显示更有趣的事情而不仅仅是简单的字符串，如自定义图形或一大段文本。在Windows Forms这种UI框架中，你需要构建自定义控件，这会带来大量的代码，并且需要维护一个新类。使用WPF内容模型就没有必要这样了。

如果不希望Content属性的值是简单的字符数组，那么就无法在控件的开放式定义中用特性来设置它。我们必须在元素的作用域内隐式定义内容数据。例如，以下的<Button>包含一个<StackPanel>作为其内容，而<StackPanel>本身包含一些数据（确切地说是一个<Ellipse>和一个<Label>）：

```
<!-- 使用复杂数据隐式设置Content属性 -->
<Button Height="80" Width="100">
  <StackPanel>
    <Ellipse Fill="Red" Width="25" Height="25"/>
    <Label Content ="OK!"/>
  </StackPanel>
</Button>
```

我们也可以使用XML的属性元素语法来设置复杂的内容。考虑如下功能等价的<Button>定义，它使用属性元素语法设置Content属性（在本章中你会发现更多有关XAML的信息，在这里不要过多关注细节）：

```
<!-- 使用属性元素语法设置Content属性 -->
<Button Height="80" Width="100">
  <Button.Content>
    <StackPanel>
      <Ellipse Fill="Red" Width="25" Height="25"/>
      <Label Content ="OK!"/>
    </StackPanel>
  </Button.Content>
</Button>
```

但请务必注意，并不是每个WPF控件都派生自ContentControl类，因此并不是所有控件都支持这种独特的内容模型（但大多数都支持）。同样，一些WPF控件改进了刚才介绍的基本内容模型。第28章将详细介绍WPF内容的作用。

2. System.Windows.Controls.Control的作用

不同于ContentControl类，所有WPF控件都共享Control基类，并将其作为公共的父类。这个基类提供了众多负责基本UI功能的核心成员。例如，Control类定义了多种属性，用于设置控件大小、透明度、焦点切换顺序逻辑、光标显示、背景颜色，等等。此外，这个父类还提供了对模板服务的支持。如第30章中所示，WPF控件可以使用模板和样式动态地改变它们的外观。表27-6按功能相关性分类，列出了Control类型的主要成员。

表27-6 Control类型的核心成员

成　　员	作　　用
Background、Foreground、BorderBrush、BorderThickness、Padding、HorizontalContentAlignment和VerticalContentAlignment	这些属性允许我们进行有关控件如何被呈现和定位的基本设置
FontFamily、FontSize、FontStretch和FontWeight	这些属性控制各种字体相关的设置
IsTabStop和TabIndex	这些属性用于创建窗口上控件之间的Tab次序
MouseDoubleClick和PreviewMouseDoubleClick	这些事件处理组件的双击行为
Template	这个属性用来获取和设置可用于改变控件渲染输出的控件模板

3. System.Windows.FrameworkElement的作用

基类提供了许多在WPF框架中都会用到的底层成员，可用于支持情节提要（在动画中使用）、数据绑定以及命名成员（通过Name属性），获取由派生类型定义的资源，创建派生类型的整体大小。表27-7列出了主要的成员。

表27-7 FrameworkElement类型的主要成员

成　　员	作　　用
ActualHeight、ActualWidth、MaxHeight、MaxWidth、MinHeight、MinWidth、Height和Width	控制派生类型的大小
ContextMenu	获取或设置和派生类型关联的弹出菜单
Cursor	获取或设置和派生类型关联的鼠标指针
HorizontalAlignment和VerticalAlignment	控制类型如何在容器中定位（如面板或列表框）
Name	允许为类型赋一个名字，这样就可以在代码文件中访问其功能
Resources	提供了对任何由类型定义的资源的访问（对于WPF资源系统的研究，见第30章）
ToolTip	获取或设置和派生类型关联的工具提示

4. System.Windows.UIElement的作用

在Window继承链的所有类型中，UIElement基类提供的功能最多。UIElement的主要任务是提供一组可以允许其子类获取焦点并处理输入请求的事件。比如，该类提供了许多负责拖放、鼠标移动、键盘输入和笔输入（为Pocket PC与Tablet PC提供的）等的操作事件。

第29章会深入研究WPF事件模型，不过你对许多核心事件应该感到很熟悉（MouseMove、KeyUp、MouseDown、MouseEnter、MouseLeave等）。除了定义许多事件之外，父类还提供了许多属性来处理控件焦点、启用状态、可见性以及命中测试逻辑，如表27-8所示。

表27-8　UIElement类型的核心成员

成　员	作　用
Focusable和IsFocused	这些属性用来设置某个派生类型的焦点
IsEnabled	这些属性用来控制某个派生类型是否启用
IsMouseDirectlyOver和IsMouseOver	这些属性提供了简单的方式来进行命中测试逻辑
IsVisible和Visibility	这些属性用来对派生类型进行可见性设置
RenderTransform	这些属性用来创建会用于渲染派生类型的转换

5. System.Windows.Media.Visual的作用

Visual类提供了WPF中的核心显示支持，包括图像数据的命中测试、坐标变换以及边框计算。事实上，Visual类与DirectX子系统交互，并在屏幕上绘制数据。第29章中会介绍，WPF提供了3个可能的形式用来渲染图形数据，每一个在功能和性能上都不一样。使用Visual类型（及其子类型，如DrawingVisual)提供了最轻量的方式来渲染图形数据，但是需要很多手动代码来处理所有必要的服务。同样，更多细节见第29章。

6. System.Windows.DependencyObject的作用

WPF支持一类称为依赖属性的特殊属性。简单地讲，这种属性风格提供额外的代码，可以使属性响应多种WPF技术，如样式、数据绑定、动画等。为了让类型支持这个新的属性架构，类型就需要从DependencyObject基类派生。虽然依赖属性是WPF开发的重点，但大多数情况下它们的细节都被隐藏了。第28章会深入讨论依赖属性。

7. System.Windows.Threading.DispatcherObject的作用

Window类型的最后一个基类（除了System.Object类，对此我认为不需要在这里给出更多的解释）是DispatcherObject。这个类型提供了一个有趣的属性Dispatcher，它可以返回相关联的System.Windows.Threading.Dispatcher对象。这个Dispatcher类是WPF应用程序事件队列的入口点，它提供了用于处理并发和线程的基本构造。

27.4　创建不使用 XAML 的 WPF 应用程序

考虑到所有的功能都是由Window类型的父类提供的，在应用程序中可以通过直接创建一个Window对象，或将其用作一个强类型子类的父类的方式来表示一个窗体。让我们在下面这段代码示例中探讨一下这两种方式。虽然大部分WPF应用程序都会使用XAML，但这完全是可选的。XAML能够表现的任何东西都完全可以通过代码来表现，反之亦然。如果你愿意，可以使用底层的对象模型和过程代码创建一个完整的WPF项目工程。

举例说明，让我们直接使用Application类和Window类而不使用XAML来创建一个最简化的完整的应用程序。先创建一个新控制台应用程序WpfAppAllCode(不必担心，本章后面将使用Visual Studio WPF项目模板)。接着，访问Project→AddReference菜单项打开AddReference对话框，并添加对WindowsBase.dll、PresentationCore.dll、System.Xaml.dll和PresentationFramework.dll的引用。

现在，用下述代码更新初始的C#文件，它将创建有适量功能的窗体（这里我只显示了必须引入来编译代码的命名空间，请自行忽略那些自动添加的using语句）：

```
// 不使用XAML写的一个简单的WPF应用程序
using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfAppAllCode
{
  // 在这第一个示例中，为了表示应用程序本身和其主窗口，我们定义了一个简单的类
  class Program : Application
  {
    [STAThread]
    static void Main(string[] args)
    {
      // 处理Startup和Exit事件，随后运行应用程序
      Program app = new Program();
      app.Startup += AppStartUp;
      app.Exit += AppExit;
      app.Run();  // 触发Startup事件
    }

    static void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }

    static void AppStartUp(object sender, StartupEventArgs e)
    {
      // 创建一个Window对象，同时设置一些基本属性
      Window mainWindow = new Window();
      mainWindow.Title = "My First WPF App!";
      mainWindow.Height = 200;
      mainWindow.Width = 300;
      mainWindow.WindowStartupLocation = WindowStartupLocation.CenterScreen;
      mainWindow.Show();
    }
  }
}
```

说明 WPF应用程序的Main()方法必须标记[STAThread]特性，这能确保应用程序所用的所有遗留COM对象都是线程安全的。否则，将会引发运行时异常。

请注意，Program类扩展了System.Windows.Application类。在这个类型的Main()方法中，我们创建了一个应用实例，并且使用方法组转换的语法处理了Startup和Exit事件。回想一下第10章所学的内容，这种简写方法无须手工指定对应的委托。当然，如果你愿意，你可以直接用名字指定对应的委托。

在下面这个经过修改的Main()方法中，请注意Startup事件被连接到了StartupEventHandler这个委托上，而这个委托只能指向以Object对象作为第一个参数，并以StartupEventArgs作为第二个参数的那些方法。另一方面，Exit事件与ExitEventHandler这个委托协作，这个委托要求其指向的方法必须以一个ExitEventArgs类型作为第二个参数。

```
[STAThread]
static void Main(string[] args)
{
  // 这一次，指定对应的委托
```

```
    Program app = new Program();
    app.Startup += new StartupEventHandler(AppStartUp);
    app.Exit += new ExitEventHandler(AppExit);
    app.Run();  // 触发Startup事件
  }
```

AppStartup()方法被配置后，用来创建Window对象，建立一些非常基本的属性设置，并调用Show()方法来在屏幕上显示这个无模式风格（ShowDialog()方法可以用来启动一个模式对话框）的窗体。而AppExit()方法只是简单地利用了WPF中的MessageBox类，在程序被终止时显示一条诊断信息。

执行这个程序，你会看到一个非常简单的主窗体，它可被最小化、最大化和关闭。为了增加一些趣味性，我们还要再加入一些用户界面元素。不过在我们这样做之前，让我们先来将代码库加以重构，使之成为一个强类型的、封装良好的Window派生类。

27.4.1　创建强类型的 Window类

目前，我们的Application派生类在应用程序启动时直接创建一个Window类型的实例。理想状况下，我们应当创建一个继承自Window类的类以封装它的外观和功能。假设我们已经在当前的WpfAppCode命名空间中创建了如下类定义（如果在新的C#文件放置这个类，一定要导入System.Windows命名空间）：

```
class MainWindow : Window
{
  public MainWindow(string windowTitle, int height, int width)
  {
    this.Title = windowTitle;
    this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
    this.Height = height;
    this.Width = width;
  }
}
```

现在我们可以更新Startup事件处理程序，简单地直接创建一个MainWindow类型的实例，如下所示：

```
static void AppStartUp(object sender, StartupEventArgs e)
{
  // 创建一个MainWindow对象
  MainWindow wnd = new MainWindow("My better WPF App!", 200, 300);
  wnd.Show();
}
```

当程序被重新编译和执行后，输出是相同的。这样做一个明显的好处就是我们现在可以在一个强类型的表示主窗体的类的基础之上开始我们的工作了。

说明　当你创建一个Window（或者派生自Window）的对象时，它将被自动加入到Application类的Windows集合中（这是通过Window类本身的一些构造函数逻辑实现的）。因此，现在可以在内存中通过Application.Windows属性迭代Window对象列表。

27.4.2　创建简单的用户界面

向C#代码中的Window添加UI元素（如Button），步骤如下所示。

(1) 定义一个成员变量来代表所需的控件；

(2) 在创建Window时配置这个控件的外观和行为；

(3) 将控件分配给继承的Content属性，或作为AddChild()方法的参数。

WPF控件的内容模型要求只能为一个元素设置Content属性。当然，只包含单一UI控件的Window是没什么用的。因此，几乎在所有情况下，分配给Content属性的“单一内容”实际上是一个布局管理器，如DockPanel、Grid、Canvas或StackPanel。在布局管理器内，可以对控件进行任意组合，甚至可以内嵌另一个布局管理器（关于WPF开发的这方面内容，详见第28章）。

现在，我们在这个Window派生类中添加一个Button控件。单击该按钮将关闭当前窗口，这会间接终止应用程序，因为内存中已经没有其他窗口了。请思考下面这段对MainWindow类的更新的代码（确保导入System.Windows.Controls以便可以访问Button类）：

```
class MainWindow : Window
{
  // 我们的UI元素
  private Button btnExitApp = new Button();

  public MainWindow(string windowTitle, int height, int width)
  {
    // 配置按钮并设置子控件
    btnExitApp.Click += new RoutedEventHandler(btnExitApp_Clicked);
    btnExitApp.Content = "Exit Application";
    btnExitApp.Height = 25;
    btnExitApp.Width = 100;

    // 将窗体的内容设置为一个按钮
    this.Content = btnExitApp;

    // 配置窗体
    this.Title = windowTitle;
    this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
    this.Height = height;
    this.Width = width;
    this.Show();
  }

  private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
  {
    // 关闭窗体
    this.Close();
  }
}
```

但是请务必注意，这个WPF按钮的Click事件与一个名为RoutedEventHandler的委托相协作，你当然会问，“什么是路由事件？”你将在第28章学习这个新的WPF事件模型。现在只需要了解RoutedEventHandler委托的目标必须提供一个object作为第一个参数，RoutedEventArgs作为第二个参数。

不管怎么样，编译并运行这个应用程序后，就可以发现如图27-6所示的自定义窗口。注意我们的按钮自动放在了窗口客户端区域的正中间，对于没有放到WPF面板类型中的内容来说，这是默认行为。



图27-6 一个用100%C#语言编写的简单的WPF应用程序

27.4.3 与应用程序级别的数据交互

回忆一下，Application类定义了一个叫Properties的属性，它允许我们通过类型索引器定义一个名称/值对的集合。因为这个索引器被定义为操作System.Object类型，所以我们可以在这个集合中保存任何项（包括自定义类），之后使用友好名来获取。使用这个方式，就可以在WPF应用程序中跨越所有窗口来共享数据。

为了演示，我们更新当前的Startup事件处理程序来检查传入的命令行参数中叫/GODMODE的值（许多PC视频游戏的作弊码）。如果我们找到了这样的标识，就在属性集合中以相同名字创建一个bool值并设置为true（否则设置值为false）。

听上去很简单，但是还有一个问题，那就是我们怎么把命令行参数（一般从Main()方法中获得）传给Startup事件处理程序呢？一个方法是调用静态的Environment.GetCommandLineArgs()方法。然而，这些参数会自动加到传入的StartupEventArgs参数中并且可以通过Args属性直接获取。那么，这里就是我们的首次更新：

```
private static void AppStartUp(object sender, StartupEventArgs e)
{
  // 检查传入的命令行参数来看是否它们指定了/GODMODE标识
  Application.Current.Properties["GodMode"] = false;
  foreach(string arg in e.Args)
  {
    if (arg.ToLower() == "/godmode")
    {
      Application.Current.Properties["GodMode"] = true;
      break;
    }
  }

  // 创建一个MainWindow对象
  MainWindow wnd = new MainWindow("My better WPF App!", 200, 300);
}
```

应用程序范围数据可以在WPF应用程序中的任何地方进行访问。我们需要做的只是（通过Application.Current）获取指向全局应用程序对象的访问点并且查看集合。例如，我们可以这样更新主窗口Button类型的Click事件处理程序：

```
private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
{
  // 用户启动了/godmode吗
  if((bool)Application.Current.Properties["GodMode"])
    MessageBox.Show("Cheater!");

  this.Close();
}
```

这样，如果用户按如下所示启动程序：

```
WpfAppAllCode.exe /godmode
```

他就会在应用程序终止时看到一个丢脸的对话框。

说明 你可以在Visual Studio中提供命令行参数。双击Solution Explorer中的Properties图标，在弹出的编辑器中单击Debug选项卡，在Command line arguments框中输入/godmode。

27.4.4 处理Window对象的关闭

终端用户可以使用许多内置的系统级别的技术来关闭窗口（如单击窗口框架的X关闭按钮），或者间接调用Close()方法来响应一些用户交互元素（如File→Exit）。不管怎样，可以拦截WPF提供的两个事件来检测用户是否真的要关闭窗口并且从内存中移除。第一个事件是Closing，它和CancelEventHandler委托一起使用。

这个委托期望目标方法接受System.ComponentModel.CancelEventArgs作为第二个参数。CancelEventArgs提供了Cancel属性，将它设置为ture可以防止窗口真正关闭（如果你问用户是否真的希望关闭窗口或需要保存它们工作的话就有用）。

如果用户确实希望关闭窗口，可以把CancelEventArgs.Cancel设置为fale（默认）。然后就会触发Closed事件（和System.EventHandler委托一起使用），这个时候窗口就会真正关闭。

向当前构造函数中添加下列代码语句来更新MainWindow类，以便处理这两个构造函数，如下所示：

```
public MainWindow(string windowTitle, int height, int width)
{
...
  this.Closing += MainWindow_Closing;
  this.Closed += MainWindow_Closed;
}
```

现在，按如下所示实现相应的事件处理程序：

```
private void MainWindow_Closing(object sender,
  System.ComponentModel.CancelEventArgs e)
{
  // 用户是否真正希望关闭这个窗口
  string msg = "Do you want to close without saving?";
  MessageBoxResult result = MessageBox.Show(msg,
    "My App", MessageBoxButton.YesNo, MessageBoxImage.Warning);

  if (result == MessageBoxResult.No)
  {
    // 如果用户不希望关闭，则取消关闭
```

27

```
            e.Cancel = true;
        }
    }
    private void MainWindow_Closed(object sender, EventArgs e)
    {
        MessageBox.Show("See ya!");
    }
```

现在运行程序，并通过单击窗口右上角的“×”图标或按钮控件来关闭窗口。你将看到如图27-7所示的确认对话框。

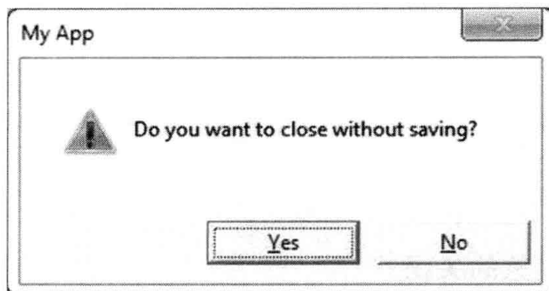


图27-7 捕获Window的Closing事件

单击Yes按钮，应用程序将被关闭。单击No按钮，窗口将仍然保留在内存中。

27.4.5 拦截鼠标事件

WPF API提供了我们可以捕获的事件来和鼠标交互。具体而言，UIElement基类定义了许多鼠标相关的事件，如MouseMove、MouseUp、MouseDown、MouseEnter、MouseLeave等。

例如，处理MouseMove事件。这个事件和System.Windows.Input.MouseEventHandler委托一起使用，它期望其目标接受System.Windows.Input.MouseEventArgs类型作为第二个参数。使用MouseEventArgs，我们可以提取出鼠标的(x, y)位置以及其他细节。考虑如下部分定义：

```
public class MouseEventArgs : InputEventArgs
{
...
  public Point GetPosition(IInputElement relativeTo);
  public MouseButtonState LeftButton { get; }
  public MouseButtonState MiddleButton { get; }
  public MouseDevice MouseDevice { get; }
  public MouseButtonState RightButton { get; }
  public StylusDevice StylusDevice { get; }
  public MouseButtonState XButton1 { get; }
  public MouseButtonState XButton2 { get; }
}
```

说明 XButton1和XButton2属性用来与“鼠标扩展按钮”交互（如某些鼠标的“next”和“previous”按钮）。它们常用于浏览器的历史列表，可以在访问过的页面之间进行导航。

GetPosition()方法允许我们获取窗口上相对UI元素的(x, y)值。如果你对捕获相对激活窗口的位置有兴趣的话，只需要传入this。在MainWindow类的构造函数中处理MouseMove事件，如下所示:

```
public MainWindow(string windowTitle, int height, int width)
{
...
  this.MouseMove += MainWindow_MouseMove;
}
```

这里是MouseMove的事件处理程序，它会在窗口的标题区域中显示鼠标的位置（注意，我们通过ToString()把返回的Point类型转换成字符串）:

```
private void MainWindow_MouseMove(object sender,
  System.Windows.Input.MouseEventArgs e)
{
  // 将窗口的标题设置为鼠标当前的(x,y)
  this.Title = e.GetPosition(this).ToString();
}
```

27.4.6 拦截键盘事件

处理当前窗体的键盘事件仍然很简单。UIElement定义了许多我们可以捕获的事件来拦截活动元素上的键盘敲击（如KeyUp、KeyDown等）。KeyUp和KeyDown事件都和System.Windows.Input.KeyEventHandler委托一起使用，它期望目标的第二个事件处理程序是KeyEventArgs类型的，它定义了几个有趣的公共属性，如下所示:

```
public class KeyEventArgs : KeyboardEventArgs
{
...
  public bool IsDown { get; }
  public bool IsRepeat { get; }
  public bool IsToggled { get; }
  public bool IsUp { get; }
  public Key Key { get; }
  public KeyStates KeyStates { get; }
  public Key SystemKey { get; }
}
```

我们使用下面的代码来演示对MainWindow构造函数中添加的KeyDown事件的处理（与前面的那些事件一样），它用当前按键的值修改按钮的内容:

```
private void MainWindow_KeyDown(object sender, System.Windows.Input.KeyEventArgs e)
{
  // 显示按钮上的按键
  btnExitApp.Content = e.Key.ToString();
}
```

双击Solution Explorer的Properties图标，在Application选项卡中将Output Type设置为Windows Application，这样可以避免启动控制台窗口。图27-8显示了第一个WPF程序的最终情况。

至此，WPF看上去只是一个新的GUI框架，它（只不过）提供了和Windows Forms、MFC或VB6相同的服务。如果真是这样的话，你肯定要质疑另一个UI工具包存在的意义。为了真正看到WPF的独特之处，需要理解新的基于XML的语法，XAML。

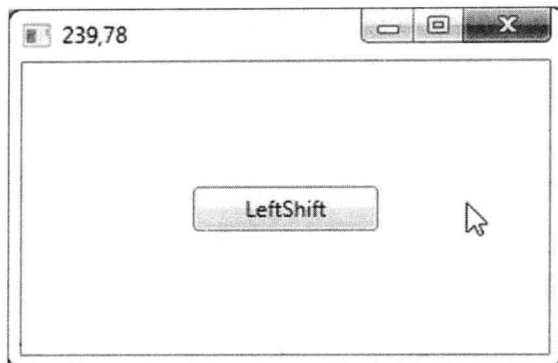


图27-8　第一个WPF程序，百分之百不含XAML

源代码　WpfAppAllCode项目的源代码位于Chapter 27子目录下。

27.5　仅使用 XAML 构建 WPF 应用程序

典型的WPF应用程序并不像第一个示例那样完全由代码组成。相反，C#代码文件将与相关的XAML源文件成对出现，它们表示给定的Window或Application的整体，以及其他还没有介绍的类型，如UserControl和Page。

这种方法称为构建WPF应用程序的代码文件法，本书其余WPF相关的部分将广泛使用这种技术。但在这之前，我们先来演示如何只用XAML文件构建WPF应用程序。尽管并不推荐这种“100% XAML”的方法，但这有助于你理解有多少标记被转换为相应的C#代码，并最终编译为.NET程序集。

说明　下面的示例会用到很多XAML技术，这将在后面正式介绍，因此如果遇到不熟悉的语法请不必担心。你可以简单地将文件加载到文本编辑器中查看，但别使用Visual Studio。下面示例中的一些XAML无法在Visual Studio可视化设计器中显示。

一般来说，XAML文件包含描述窗口外观的标记，而C#代码文件包含实现逻辑。例如，Window的XAML文件会描述整个布局系统及其内部的控件，指定不同事件处理程序的名称。而相关的C#文件则包含这些事件处理程序的实现逻辑和应用程序所需的自定义代码。

XAML（Extensible Application Markup Language，可扩展应用程序标记语言）是一套新的基于XML的语法，它允许你通过标记来定义一个.NET类型的状态（也可以是功能）。虽然XAML往往用于为WPF创建UI，事实上它可以用来描述任何非抽象的.NET类型树（包括你在自定义的.NET程序集中定义的自定义类型），前提是每个类型都提供一个默认的构造函数。你将会看到，在一个*.xaml文件中建立的标记会被转化为一个直接映射到相关的.NET命名空间中的一个完整的对象模型。

因为XAML是基于XML的语法，我们获得了XML带给我们的所有好处与坏处。从好的方面来说，

XAML文件具有非常好的自描述性（就像任何XML文档一样）。一般来讲，XAML文件中的每个元素都代表某个.NET命名空间中的一个类型名称（例如Button、Window或者Application）。一个起始元素范围内的特性（一般说来）将映射到这个特定类型的特性（Height、Width等）和事件（Startup、Click等）。

由于XAML只是定义对象状态的一种声明方式，还可以通过标记或程序代码来定义一个WPF组件。例如，如下所示的XAML：

```
<!-- 使用XAML定义一个WPF按钮 -->
<Button Name = "btnClickMe" Height = "40" Width = "100" Content = "Click Me" />
```

可用如下的编程方式来表示：

```
// 定义原来那个在C#代码中定义过的WPF按钮
Button btnClickMe = new Button();
btnClickMe.Height = 40;
btnClickMe.Width = 100;
btnClickMe.Content = "Click Me";
```

从坏的方面来说，XAML比较冗长并且区分大小写（跟XML文档一样）。同样，复杂的XAML定义将产生大量的标记。大多数开发人员都不必手工编写WPF应用程序的完整XAML描述。而这个工作的绝大部分（幸好是绝大部分）将交给开发工具来完成，例如Visual Studio、Microsoft Expression Blend，或者一些第三方产品。当这些工具生成基本的标记之后，你就可以根据需要对XAML定义进行细调了。

27.5.1 用XAML定义窗体对象

虽然工具可以为你生成大量的XAML，但了解XAML语法的基本工作原理以及这些标记最终如何被转化为一个有效的.NET程序集对你来讲非常重要。为了说明XAML语法的基础，我们的下一个示例将只使用一组.xaml文件来实现前面的WPF应用程序。

目前，我们的MainWindow是在C#中定义为一个扩展了System.Windows.Window基类的类。这个类包含了一个Button对象，在它被单击时会调用一个注册了的事件处理程序。用XAML语法定义同样的Window类型可由如下代码完成。首先，使用简单的文本编辑器（如Notepad）新建一个MainWindow.xaml文件，并将其保存在从C:盘可以很容易访问的子目录中，因为我们会通过命令行处理该文件。现在添加如下的XAML：

```
<!-- 这里是我们的窗体定义 -->
<Window x:Class="WpfAppAllXaml.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="A Window built using 100% XAML"
  Height="200" Width="300"
  WindowStartupLocation ="CenterScreen">

  <!-- 该窗体以单个按钮作为其内容 -->
  <Button x:Name="btnExitApp" Width="133" Height="24"
         Content = "Close Window" Click ="btnExitApp_Clicked"/>

  <!-- 按钮的Click事件处理方法的实现 -->
  <x:Code>
    <![CDATA[
      private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
```

```
      {
        this.Close();
      }
    ]]>
  </x:Code>
</Window>
```

首先，注意根元素<Window>使用了Class特性指定C#类的名称，在处理该XAML文件时将自动生成这个C#类。同时还要注意Class特性的前缀为x:标签。在<Window>开放元素内，这个XML标签前缀被设置为字符串“http://schemas.microsoft.com/winfx/2006/xaml”来建立一个XML命名空间声明。本章稍后将详细介绍XML命名空间的定义，此时只需要注意在引用“http://schemas. microsoft.com/winfx/2006/xaml”命名空间定义的项时，必须使用x:前缀。

在起始标签<Window>的范围内，我们已经为Title、Height、Width和WindowStartupLocation特性赋了值，你可以看到这是对PresentationFramework.dll程序集中System.Windows.Window类提供的同名属性的一个直接映射。

接下来，请注意在窗体定义的范围内，我们创建了描述Button对象的外观和行为的标记。它将被用于设置窗体的Content属性。除了设置变量名（使用x:Name XAML标记）和总的尺寸以外，我们还处理了Button类型的Click事件，设置了Click事件发生时的委托处理方法。

XAML文件最后一部分内容是<x:Code>元素，毫不奇怪，它允许我们在一个*.xaml文件中直接编写事件处理方法和该类的其他方法。作为一种安全措施，代码本身被封装在一个CDATA范围内，以防止XML解析器试图直接解析这些数据（对于现在这个示例而言，这不是必需的）。

有必要说明一下，不推荐在<Code>元素中编写功能函数。虽然这种“单一文件方式”将所有的行为都隔离在一个地方，但内嵌代码并没有为我们提供标记与逻辑关注点之间的清晰分离。在大多数WPF应用程序中，“真正的代码”都会放在一个相关联的部分C#类中（我们最后会这样做）。

27.5.2　用XAML定义应用对象

请记住XAML可以用来通过标记来定义任何非抽象的提供默认构造函数的.NET类。因此，我们当然也可以用标记来定义应用对象。参考下面这段新文件MyApp.xaml中的内容：

```
<!-- 看起来main()方法不见了！不过StartupUri特性在功能上是等价的 -->
<Application x:Class="WpfAppAllXaml.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
</Application>
```

你可能会同意，这里的Application继承类与其XAML描述之间的映射并不像MainWindow类与其XAML描述间的映射那么清晰。具体看，似乎没有任何Main()方法的踪迹。考虑到任何.NET程序都必须要有一个程序入口点，你可能会假设这个入口点会基于StartupUri属性部分，在编译时被创建，你是正确的。赋给StartUpUri的值表示应用程序启动时加载哪个XAML资源。在本例中，我们将StartupUri设置为定义初始窗体对象的XAML资源的名称MainWindow.xaml。

虽然Main()方法会在编译时被自动创建，如果我们愿意，我们可以使用<x:Code>元素来捕获其他C#代码块。例如，如果想在程序关闭时显示一条消息，可以处理Exit事件并按如下所示实现它（注意开放的<Application>元素现在设置了Exit特性，来捕获Application类的Exit事件）：

```
<Application x:Class="SimpleXamlApp.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml" Exit ="AppExit">
  <x:Code>
    <![CDATA[
      private void AppExit(object sender, ExitEventArgs e)
      {
        MessageBox.Show("App has exited");
      }
    ]]>
  </x:Code>
</Application>
```

27.5.3 通过msbuild.exe处理XAML文件

到现在，我们已经准备好将我们的标记转换为一个有效的.NET程序集。但如果要这样做，我们还不能直接利用C#编译器。到目前为止，C#编译器还不能直接理解XAML标记。不过，命令行工具msbuild.exe懂得如何将XAML转换为C#代码并且在获知正确的*.targets文件后，在运行时编译这些代码。

msbuild.exe工具根据基于XML构建的脚本中的指令来编译.NET代码。这些构建脚本文件所包含的数据与Visual Studio生成的*.csproj文件中的内容完全相同。因此，.NET程序可以用msbuild.exe在命令行中编译，也可以用Visual Studio本身。

说明 对msbuild.exe工具的完整介绍超出了本章的范围。要了解更多内容，可以在.NET Framework 4.5 SDK文档中搜索“MSBuild”话题。

27

以下是一个非常简单的构建脚本WpfAppAllXaml.csproj，它仅包含足够的信息供msbuild.exe将XAML文件转换为相关的C#代码库：

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>WpfAppAllXaml</RootNamespace>
    <AssemblyName>WpfAppAllXaml</AssemblyName>
    <OutputType>winexe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <ApplicationDefinition Include="MyApp.xaml" />
    <Page Include="MainWindow.xaml" />
  </ItemGroup>
  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
  <Import Project="$(MSBuildBinPath)\Microsoft.WinFX.targets" />
</Project>
```

说明　该*.csproj文件不能直接被Visual Studio加载，因为它只包含用于在命令行构建应用程序的最少指令。

这里的<PropertyGroup>元素用来指定编译中的一些基本属性，例如根命名空间、最终生成的程序集的名称以及输出类型（等价于csc.exe的选项/target:winexe）。

第一个<ItemGroup>指定了当前编译过程中要引用的外部程序集集合。可以看到，它们就是本章前面探讨过的那些WPF核心程序集。

第二个<ItemGroup>则有趣得多。请注意<ApplicationDefinition>元素的Include特性指向定义应用对象的*.xaml文件。<Page>的Include特性则可以用来列举余下的每个*.xaml文件，这些*.xaml文件定义了由该应用对象所处理的窗体（还有页面，它们经常被用于构建XAML浏览器程序）。

不过，这个构建文件的奥秘却在于最后的那些<Import>子元素。请注意我们的编译脚本引用了两个*.targets文件，每一个都包含了编译过程中的许多其他指令。Microsoft.WinFX.targets文件包含了将XAML定义转换为等价C#代码文件所需的那些编译设置，而Microsoft.CSharp.targets文件则包含了与C#编译器交互的数据。

不管怎样，你现在可以打开Developer Command Prompt，使用msbuild.exe来处理这些XAML数据。定位到MainWindow.xaml、MyApp.xaml和WpfAppAllXaml.csproj文件所在的目录，输入下面的命令：

```
msbuild WpfAppAllXaml.csproj
```

构建成功之后，你会在工作目录中看到\bin和\obj子目录（与Visual Studio项目一样）。打开\bin\Debug文件夹，会看到一个新的.NET程序集SimpleXamlApp.exe。在ildasm.exe中打开该程序集，可以看到XAML已经被转换为有效的可执行应用程序（如图27-9所示）。

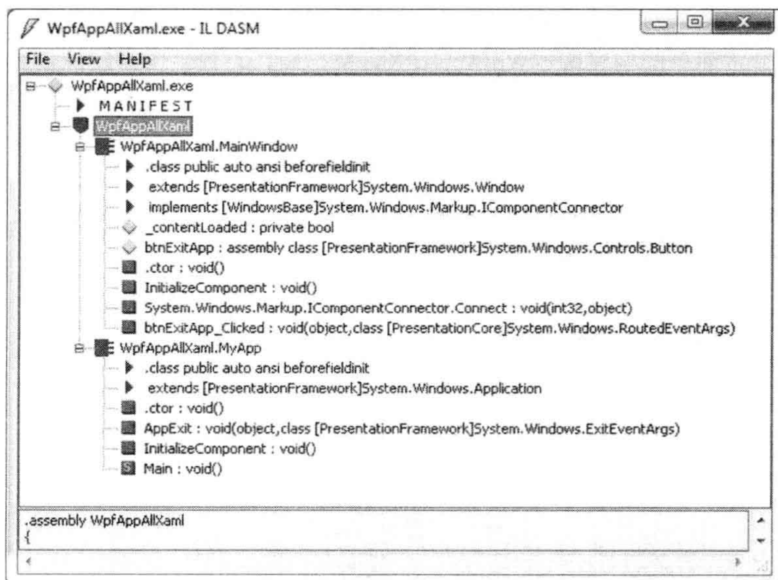


图27-9　将XAML标记转换成一个.NET可执行文件？很有意思

如果双击可执行程序来运行程序，将在屏幕上看到主窗口。

27.6　将标记转换为.NET 程序集

为了准确地理解我们的标记是如何被转换为.NET程序集的，我们需要深入一点了解msbuild.exe的处理过程，并研究一些编译器生成的文件，包括在编译时嵌入到程序集中的特定二进制资源。首先，学习一下*.xaml文件是如何转换成相应的C#代码库的。

27.6.1　将窗口XAML标记映射到C#代码

在一个msbuild脚本中指定的那些*.targets文件包含许多将XAML元素翻译为C#代码的指令。当msbuild.exe处理我们的*.csproj文件时，它以*.g.cs（这里的g代表自动生成的意思）的形式生成两个文件，它们被保存在\obj\Debug目录下。根据*.xaml文件的名字，生成的C#文件也就是MainWindow.g.cs和MyApp.g.cs。

如果打开文件MainWindow.g.cs，你会找到那个扩展了Window基类的MainWindow类。该名称与<Window>开始标记中x:Class特性的值相同。同时，这个类还定义了一个System.Windows.Controls.Button的成员变量btnExitApp。在本例中，控件的名称基于<Button>开放声明中x:Name特性的值。该类还包含按钮的Click事件的处理程序btnExitApp_Clicked()。以下是这个编译器生成的MainWindow.g.cs文件的部分代码：

```
public partial class MainWindow :
  System.Windows.Window, System.Windows.Markup.IComponentConnector
{
  internal System.Windows.Controls.Button btnExitApp;

  private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
  {
    this.Close();
  }
...
}
```

这个类定义了一个私有的bool类型的私有成员变量（名为_contentLoaded），它并不直接对应于XAML标记。该数据成员用来判断（和确保）窗口的内容只能设置一次：

```
public partial class MainWindow :
  System.Windows.Window, System.Windows.Markup.IComponentConnector
{
  // 很快会解释这个成员变量
  private bool _contentLoaded;
...
}
```

注意，编译器生成的类显式地实现了定义在System.Windows.Markup命名空间中的WPF IComponentConnector接口。这个接口定义了一个方法Connect()，它已被实现，用来构建在原始的Main Window.xaml文件中定义的事件逻辑。在该方法完成前，_contentLoaded成员变量被设为ture。下面是该方法的关键部分：

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object target)
```

```
{
  switch (connectionId)
  {
    case 1:
      this.btnExitApp = ((System.Windows.Controls.Button)(target));
      this.btnExitApp.Click += new
        System.Windows.RoutedEventHandler(this.btnExitApp_Clicked);
    return;
  }
  this._contentLoaded = true;
}
```

最后，MainWindow类也定义并实现了一个名为InitializeComponent()的方法。你可能会认为该方法会设置不同的属性（Height、Width、Content等）来建立各个控件的外观。然而事实并非如此。那么，这些控件是如何正确地放置在UI上的呢？InitializeComponent()中的逻辑定位了一个内嵌的程序集资源，它的名称与初始的*.xaml文件相同，如下所示：

```
public void InitializeComponent()
{
  if (_contentLoaded)
  {
    return;
  }
  _contentLoaded = true;
  System.Uri resourceLocater = new
    System.Uri("/WpfAppAllXaml;component/mainwindow.xaml",
           System.UriKind.Relative);
  System.Windows.Application.LoadComponent(this, resourceLocater);
}
```

至此，问题变成了：这个内嵌的资源到底是什么？

27.6.2 BAML 的作用

当msbuild.exe处理*.csproj文件时，它还生成了一个以*.baml为扩展名的文件。它是根据初始的MainWindow.xaml的文件名来命名的，因此你应该能在\obj\Debug文件夹下看到文件MainWindow.baml（如图27-10所示）。

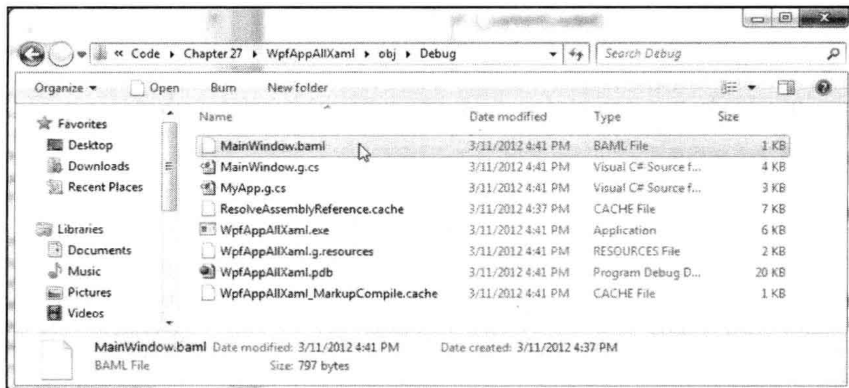


图27-10　BAML只是XAML的简洁二进制版本

可能你已经从它的名字中猜到了，二进制应用程序标记语言（BAML）是XAML的一种二进制表示。这个*.baml文件作为资源（通过一个生成的*.g.resources文件）内嵌在编译好的程序集中。

BAML资源包含用于建立UI部件外观所需的所有数据（同样，例如窗体的`Height`和`Width`属性）。事实上，如果通过Visual Studio打开*.baml文件，你就能看到初始的XAML特性的踪迹（见图27-11）。

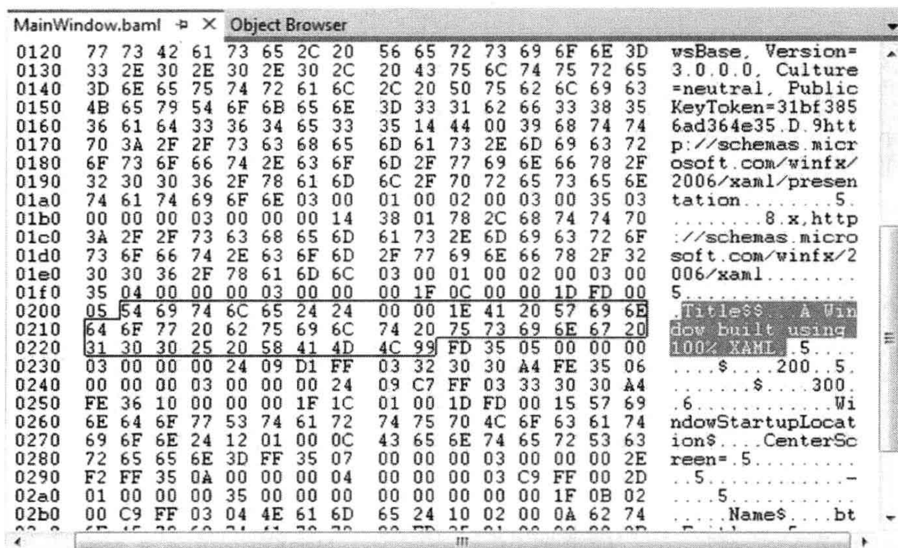


图27-11 BAML包含用于在运行时构建对象的属性值

这里最重要的是要理解WPF应用程序本身包含标记的二进制表示（BAML）。在运行时，BAML从资源容器中被取出，并用来确保所有窗口和控件的外观都能正确初始化。

同样还要记住，该二进制资源的名称与你编写的独立的*.xaml文件的名称完全相同。但这并不意味着必须将松散的*.xaml文件随编译的WPF程序一同发布。除非WPF应用程序要在运行时动态加载和解析*.xaml文件，否则永远没有必要发布原始的标记。

27.6.3 将应用程序XAML标记映射到C#代码

自动生成的代码谜题的最后一关出现在MyApp.g.cs文件中。在这里，我们看到了派生自`Application`的类有一个适当的`Main()`入口点方法。这个方法的实现调用了这个`Application`派生类的`InitializeComponent()`方法，它反过来又为`StartupUri`属性赋了值，从而使得每个对象都可以根据二进制的XAML定义来为其自身建立正确的属性设置。

```
namespace WpfAppAllXaml
{
  public partial class MyApp : System.Windows.Application
  {
    void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }
```

```
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public void InitializeComponent()
    {
      this.Exit += new System.Windows.ExitEventHandler(this.AppExit);
      this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);
    }

    [System.STAThreadAttribute()]
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    public static void Main() {
      SimpleXamlApp.MyApp app = new SimpleXamlApp.MyApp();
      app.InitializeComponent();
      app.Run();
    }
  }
}
```

27.6.4 XAML到程序集的过程摘要

本章到此为止，我们只用了两个XAML文件和一个相关的构建脚本创建了一个完整的WPF程序。可以看到，在编译的过程中，msbuild.exe使用了定义在*.targets文件中的辅助设置来处理这些XAML文件（并生成*.baml）。图27-12对编译期处理*.xaml文件的过程做了全程图解。

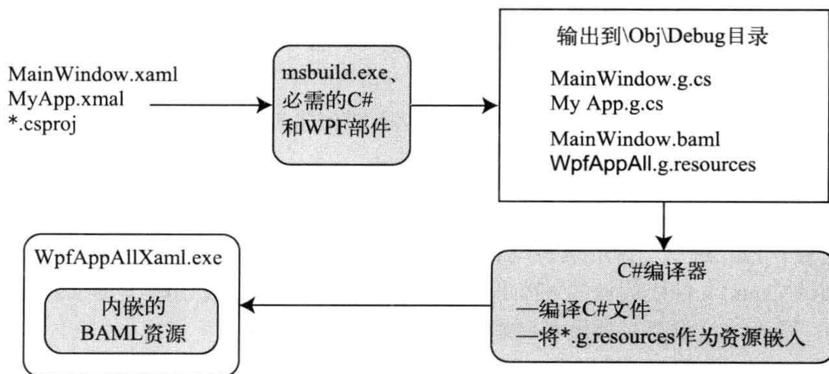


图 27-12 XAML 到程序集的编译期处理过程

希望你已经理解了如何使用XAML数据来构建.NET应用程序。接下来我们来看一下XAML的语法和语义。

源代码 WpfAppAllXaml项目的源代码位于Chapter 27子目录下。

27.7 WPF XAML 语法

产品级的WPF应用程序通常会使用专门的工具来生成必要的XAML。理解XAML标记的整个结构

与这些工具一样，是十分有用的。为了帮助我们的学习过程，我将介绍一个十分流行（并且免费）的工具，可以用来简单地体验XAML。

27.7.1 Kaxaml

在首次学习XAML语法时，使用免费工具Kaxaml是十分有帮助的。这个流行的XAML编辑器/解析器可以在网站http://www.kaxaml.com下载。

Kaxaml是非常有帮助的，它不关心C#源代码、事件处理程序或实现逻辑，使用它测试XAML片段比使用Visual Studio WPF项目模板要更加简单直接。同样，Kaxaml还包含很多集成工具，如颜色选择器、XAML片段管理器，甚至还包括根据设置格式化XAML的“XAML清理器”。首次打开Kaxaml时，将看到一个简单的<Page>控件标记，如下所示：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>

  </Grid>
</Page>
```

与Window类似，Page也包含不同的布局管理器和控件。但与Window不同的是，Page对象不能作为独立的实体运行，它们必须被放置在合适的宿主中，如NavigationWindow、Frame或Web浏览器（此时创建的是一个XBAP）。不过在<Page>和<Window>的作用域内，我们可以输入相同的标记。

说明 在Kaxaml标记窗口内将<Page>和</Page>元素更改为<Window>和</Window>，按下F5键即可在屏幕上加载新的窗口。

在该工具下方的XAML面板内输入如下标记，来进行最初的测试：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- 包含自定义内容的按钮 -->
    <Button Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

你可以在Kaxaml编辑器上方看到所呈现的页面（如图27-13所示）。

使用Kaxaml时要记住，它不允许你编写任何需要代码编译的标记（但可以使用x:Name），包括定义x:Class特性（为指定代码文件）、在标记中输入事件处理程序的名称以及使用其他需要代码编译的XAML关键字（如FieldModifier或ClassModifier）。否则，将导致标记错误。

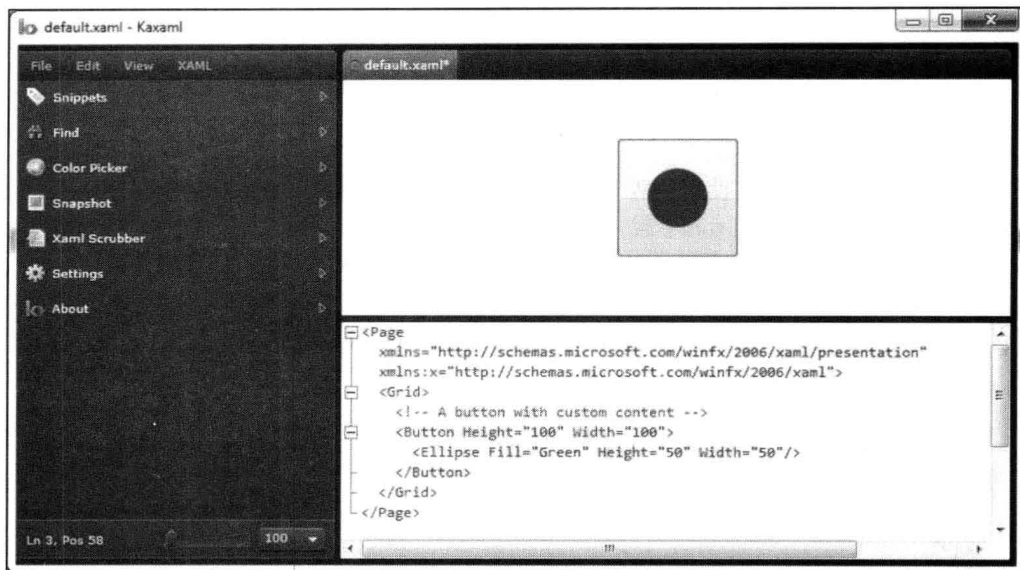


图27-13　Kaxaml是学习XAML语法非常有帮助（且免费）的工具

27.7.2　XAML XML 命名空间和 XAML 关键字

WPF相关的XAML文件的根元素（如<Window>、<Page>、<UserControl>或<Application>定义）一般定义为引用两个XML命名空间：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>

  </Grid>
</Page>
```

第一个XML命名空间http://schemas.microsoft.com/winfx/2006/xaml/presentation映射许多当前*.xaml文件使用的WPF .NET命名空间(System.Windows、System.Windows.Controls、System.Windows.Data、System.Windows.Ink、System.Windows.Media和System.Windows.Navigation等)。

这个一对多的映射其实使用程序集级别的[XmlnsDefinition]特性硬编码在WPF程序集中（WindowsBase.dll、PresentationCore.dll和PresentationFramework.dll）。例如，打开Visual Studio对象浏览器并选择PresentationCore.dll程序集，会看到下面的列表，它实际上导入了System.Windows命名空间：

```
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml/presentation",
                           "System.Windows")]
```

第二个XML命名空间http://schemas.microsoft.com/winfx/2006/xaml用于包含XAML特定的关键字和System.Windows.Markup命名空间中的类型的子集，如下所示：

```
[assembly: XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml",
                           "System.Windows.Markup")]
```

格式明确的XML文档（记住，XAML是基于XML的语法）必须定义一个根元素来指定一个XML命名空间作为主命名空间，它一般是包含大多数公共使用项的命名空间。如果根元素需要包含其他二级命名空间（如这里），就必须使用唯一前缀来定义（来解决可能的命名冲突）。作为惯例，前缀只是x，然而也可以是我们需要的特殊表示，如XamlSpecificStuff：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:XamlSpecificStuff="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- 具有自定义内容的按钮 -->
    <Button XamlSpecificStuff:Name="button1" Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

定义这么长的XML命名空间前缀的明显缺点是，每次XAML文件需要引用定义在这个XML命名空间中的类型时，就需要写XamlSpecificStuff。由于XamlSpecificStuff需要许多额外的键盘敲击，我们还是使用x吧。

不管怎样，除了x:Name、x:Class和x:Code关键字以外，http://schemas.microsoft.com/winfx/2006/xaml XML命名空间还提供了对其他XAML关键字的访问，表27-9列出了一些主要的关键字。

表27-9 XAML关键字

XAML关键字	作　用
x:Array	在XAML中表示.NET数组类型
x:ClassModifier	用来定义由Class关键字定义的类类型的可见性（内部的或公开的）
x:FieldModifier	用来为任何根的命名子元素（如<Window>元素中的<Button>）定义类型成员的可见性（内部的、公开的、私有的或受保护的）。命名元素是使用XAML Name关键字定义的元素
x:Key	用来为要放到字典元素中的XAML项创建一个键值
x:Name	用来为给定的XAML元素指定生成的C#名称
x:Null	表示null引用
x:Static	用来引用类型的静态成员
x:Type	C# typeof操作符的XAML等价形式（它会根据提供的名字产生System.Type）
x:TypeArguments	用来使用指定的类型参数以泛型类型创建元素

除了这两个必要的XML命名空间的声明，还有可能（有时会必须）在XAML文档的开放元素内定义其他标签前缀。当你需要在XAML中描述外部程序集定义的.NET类时，通常需要这么做。

例如，你构建了一些自定义WPF控件，并且将它们打包到一个MyControls.dll库中。现在，如果你希望创建使用这些控件的Window，可以使用clr-namespace和assembly标记建立一个自定义的XML命名空间，使其映射到自定义库。以下是一些示例标记，创建了一个myCtrls标签前缀，可用来访问库中的控件：

```
<Window x:Class="WpfApplication1.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:myCtrls="clr-namespace:MyControls;assembly=MyControls"
  Title="MainWindow" Height="350" Width="525">
```

27

```
  <Grid>
    <myCtrls:MyCustomControl />
  </Grid>
</Window>
```

clr-namespace标记用来设置程序集中.NET命名空间的名称，assembly标记用来设置外部*.dll程序集的友好名称。你可以对任何希望在标记中进行操作的外部.NET库使用该语法。虽然此时不必这么做，但后续的章节会要求我们定义自定义XML命名空间声明，来在标记中描述类型。

说明 如果要在标记中定义的类属于当前程序集，但位于不同的.NET命名空间，那么xmlns标签后面就不需要定义assembly=特性。例如：

```
xmlns:myCtrls="clr-namespace:SomeNamespaceInMyApp"
```

27.7.3 控制类和成员变量的可见性

在必要的时候，我们会演示这些关键字，作为一个简单的示例，考虑如下XAML <Window>定义，它使用了ClassModifier和FieldModifier关键字以及x:Name和x:Class（要记住，kaxaml.exe不允许使用任何会产生代码编译的XAML关键字，如x:Code、x:FieldModifier或x:ClassModifier）：

```
<!-- 这个类在*.g.cs文件中被声明为内部的 -->
<Window x:Class="MyWPFApp.MainWindow" x:ClassModifier ="internal"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <!-- 这个按钮在*.g.cs文件中会是公开的 -->
  <Button x:Name ="myButton" x:FieldModifier ="public" Content = "OK"/>
</Window>
```

默认情况下，所有C#/XAML类型定义都是公开的，而成员默认都是内部的。然而，根据我们的XAML定义，最后自动生成的文件会包含一个具有一个公开Button变量的内部类类型：

```
internal partial class MainWindow : System.Windows.Window,
       System.Windows.Markup.IComponentConnector
{
       public System.Windows.Controls.Button myButton;
       ...
}
```

27.7.4 XAML元素、XAML特性和类型转换器

创建了根元素和任何必需的XML命名空间之后，下一个任务就是为根元素填充子元素。在一个真实的WPF应用程序中，子元素可能是一个布局管理器（如Grid或StackPanel），它包含许多描述用户界面的其他元素。下一章会详细研究这些布局管理器，现在假设我们的<Window>类型会包含一个Button元素。

在本章中你已经看到了，XAML元素映射到某个.NET命名空间中的类或结构类型，而开始元素标签中的特性映射到类型的属性或事件。为了演示，在Kaxaml中输入如下<Button>定义：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- 配置按钮的外观 -->
    <Button Height="50" Width="100" Content="OK!"
            FontSize="20" Background="Green" Foreground="Yellow"/>
  </Grid>
</Page>
```

注意，为每个属性设置的值都是简单的文本值。这看上去似乎完全不匹配，因为如果在C#代码中设置Button的这些属性，就不能使用这些字符串对象，而必须使用指定的数据类型。例如，下面的代码包含了一个相同的按钮：

```
public void MakeAButton()
{
  Button myBtn = new Button();
  myBtn.Height = 50;
  myBtn.Width = 100;
  myBtn.FontSize = 20;
  myBtn.Content = "OK!";
  myBtn.Background = new SolidColorBrush(Colors.Green);
  myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

事实上，WPF发布了很多类型转换器类，可以将简单的文本值转换为正确的数据类型。这个处理过程是透明的（也是自动的）。

尽管这些已经很好了，但很多时候你需要为XAML特性设置比简单字符串要更复杂的值。例如，假设你要为Button的Background属性设置一个自定义画刷。如果在代码中构建画刷，则相当容易，如下所示：

```
public void MakeAButton()
{
...
  // 用来设置背景的画刷
  LinearGradientBrush fancyBruch =
    new LinearGradientBrush(Colors.DarkGreen, Colors.LightGreen, 45);
  myBtn.Background = fancyBruch;
  myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

但是，你能用字符串表示这个复杂的画刷吗？当然不行。幸好，XAML提供了一种特殊的语法——属性元素语法，可以将属性值设置为复杂的对象。

27.7.5 XAML属性元素语法

属性元素语法允许为属性赋予复杂的对象。这里是按钮的一个XAML描述，它使用LinearGradientBrush来设置Background属性：

```
<Button Height="50" Width="100" Content="OK!"
        FontSize="20" Foreground="Yellow">
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color="DarkGreen" Offset="0"/>
      <GradientStop Color="LightGreen" Offset="1"/>
```

```
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

注意，在<Button>和</Button>标记的作用域内，我们定义了一个子作用域<Button.Background>。在该作用域内，定义了一个<LinearGradientBrush>（不用担心画刷的具体代码，我们将在第29章中学习WPF图形）。

一般来说，任何属性都能用属性元素语法进行设置，它们总是可以分解为以下模式：

```
<DefiningClass>
  <DefiningClass.PropertyOnDefiningClass>
    <!-- 属性的值 -->
  </DefiningClass.PropertyOnDefiningClass>
</DefiningClass>
```

尽管任何属性都能使用这种语法，但如果仅仅是要捕获简单字符串的值，这样就会浪费输入时间。例如，像下面这样设置Button的Width显得有点啰嗦：

```
<Button Height="50" Content="OK!"
        FontSize="20" Foreground="Yellow">
...
  <Button.Width>
    100
  </Button.Width>
</Button>
```

27.7.6　XAML附加属性

除了属性元素语法以外，XAML还定义了特殊语法用来定义附加属性。从根本上说，附加属性允许子元素为定义在父元素中的属性定义唯一值。模板通常如下所示：

```
<ParentElement>
  <ChildElement ParentElement.PropertyOnParent = "Value">
</ParentElement>
```

附加属性最常见的用处就是在WPF布局管理器类（Grid、DockPanel等）中定位UI元素。下一章会深入这些面板的细节，下面在Kaxaml中输入如下代码：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Canvas Height="200" Width="200" Background="LightBlue">
    <Ellipse Canvas.Top="40" Canvas.Left="40" Height="20" Width="20" Fill="DarkBlue"/>
  </Canvas>
</Page>
```

我们在这里定义了一个包含Ellipse的Canvas布局管理器。注意，Ellipse使用了附加属性语法，通知父元素（Canvas）它距离左上角的位置。

关于附加属性，有几个注意事项。首先也是最重要的，它并不是一个全功能的语法，不能用于所有父类型的所有属性上。例如，下面的XAML不能通过解析：

```
<!-- 错误！不能用附加属性设置Canvas的Background属性 -->
<Canvas Height="200" Width="200">
  <Ellipse Canvas.Background="LightBlue"
           Canvas.Top="40" Canvas.Left="90"
```

```
            Height="20" Width="20" Fill="DarkBlue"/>
</Canvas>
```

其实，WPF包含一种特定概念，称为依赖属性，附加属性是它的特殊形式。除非属性以某种特殊的方式实现，否则不能对其使用附加属性语法。我们将在第31章中详细介绍依赖属性。

说明 Kaxaml、Visual Studio、Expression Blend都包含智能感知，可以显示给定元素中有效的依赖属性。

27.7.7 XAML标记扩展

如前所述，属性值通常表示为简单字符串或属性元素语法。另一种指定XAML特性值的方法是使用标记扩展。它允许XAML解析器从专门的外部类中获取属性的值。这是非常有好处的，因为有些属性需要一些代码语句来计算其值。

标记扩展使用新的功能来干净地扩展XAML的语法。它由MarkupExtension的派生类来表示。注意，你几乎不需要构建自定义标记扩展。但实际上很多XAML关键字（如x:Array、x:Null、x:Static、x:Type）都是变相的标记扩展。

标记扩展包含在大括号内，如：

```
<Element PropertyToSet = "{MarkUpExtension}"/>
```

为了学习标记扩展，在Kaxaml中编写如下标记：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:CorLib="clr-namespace:System;assembly=mscorlib">

  <StackPanel>
    <!-- Static标记扩展从类的静态成员中获取值 -->
    <Label Content ="{x:Static CorLib:Environment.OSVersion}"/>
    <Label Content ="{x:Static CorLib:Environment.ProcessorCount}"/>

    <!-- Type标记扩展是C# typeof操作符的XAML版本 -->
    <Label Content ="{x:Type Button}" />
    <Label Content ="{x:Type CorLib:Boolean}" />

    <!-- 用字符串数组填充ListBox -->
    <ListBox Width="200" Height="50">
      <ListBox.ItemsSource>
        <x:Array Type="CorLib:String">
          <CorLib:String>Sun Kil Moon</CorLib:String>
          <CorLib:String>Red House Painters</CorLib:String>
          <CorLib:String>Besnard Lakes</CorLib:String>
        </x:Array>
      </ListBox.ItemsSource>
    </ListBox>
  </StackPanel>
</Page>
```

首先，注意<Page>定义了新的XML命名空间声明，可以访问mscorlib.dll中的System命名空间。有了这个XML命名空间，我们先使用x:Static标记扩展，并获取System.Environment类的OSVersion和ProcessorCount的值。

x:Type标记扩展可以用来访问指定项的元数据描述。这里，我们简单输出WPF Button和System.Boolean类型的完全限定名。

最有趣的部分是ListBox。我们用一个完全声明在标记内部的字符串数组来设置ItemSource属性。注意x:Array标记扩展是如何在作用域内指定多个子项的：

```
<x:Array Type="CorLib:String">
  <CorLib:String>Sun Kil Moon</CorLib:String>
  <CorLib:String>Red House Painters</CorLib:String>
  <CorLib:String>Besnard Lakes</CorLib:String>
</x:Array>
```

说明 上面的XAML示例仅仅为了演示标记扩展。你将在第28章中看到更简单的生成ListBox控件的方式。

图27-14显示了Kaxaml中的<Page>标记。

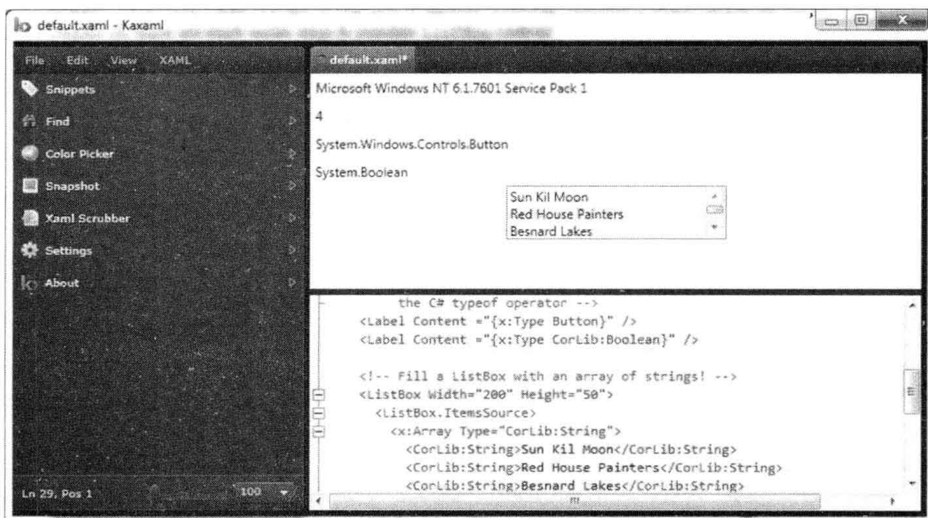


图27-14 扩展标记可以通过专门的类来设置值

此时，你已经看到了很多展示XAML语法各个核心部分的示例。你肯定会同意XAML是如此有趣，它能使你以声明的方式描述.NET对象树。这在配置图形用户界面时十分有用，但要记住，XAML可以描述任何程序集中的任何类型，只要它是一个非抽象类型，并且包含默认构造函数。

27.8 使用代码隐藏文件构建 WPF 应用程序

本章的前两个示例演示了构建WPF应用程序的极端情况，仅使用代码或仅使用XAML。但推荐的构建WPF应用的方式是使用代码文件。在这种模型下，项目中的XAML文件只包含描述类一般状态的标记，而代码文件包含了实现的细节。

27.8.1 为MainWindow类添加代码文件

为了演示这种方法，我们让WpfAppAllXaml示例使用代码文件。复制整个文件夹，并改名为WpfAppCodeFiles。现在，在该目录下创建新的C#代码文件MainWindow.xaml.cs（按照惯例，C#代码隐藏文件的名称为*.xaml.cs的形式）。在文件中添加如下代码：

```
// MainWindow.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfAppAllXaml
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      // 记住，该方法定义在自动生成的MainWindow.g.cs文件中
      InitializeComponent();
    }

    private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
    {
      this.Close();
    }
  }
}
```

这里，我们定义了一个包含事件处理逻辑的分部类，它将与*.g.cs文件中定义的相同类型的分部类进行合并。由于InitializeComponent()定义在MainWindow.g.cs文件中，窗口的构造函数调用该方法来加载和处理内嵌的BAML资源。

MainWindow.xaml文件也需要更新。要取出前面示例中的所有C#代码：

```
<Window x:Class="WpfAppAllXaml.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="A Window built using Code Files!"
  Height="200" Width="300"
  WindowStartupLocation ="CenterScreen">

  <!-- 事件处理程序现在位于代码文件中 -->
  <Button x:Name="btnExitApp" Width="133" Height="24"
          Content = "Close Window" Click ="btnExitApp_Clicked"/>
</Window>
```

27.8.2 为MyApp类添加代码文件

如果需要，也可以为Application的派生类型构建代码隐藏文件。由于大多数行为发生在MyApp.g.cs文件中，因此MyApp.xaml.cs中也不过是下面这样：

```
// MyApp.xaml.cs
using System;
using System.Windows;
```

```
using System.Windows.Controls;

namespace WpfAppAllXaml
{
  public partial class MyApp : Application
  {
    private void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }
  }
}
```

MyApp.xaml文件现在是这样的：

```
<Application x:Class="WpfAppAllXaml.MyApp"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml"
  Exit ="AppExit">
</Application>
```

27.8.3　用msbuild.exe处理代码文件

在使用msbuild.exe重新编译文件之前，需要更新*.csproj文件，使用<Compile>元素（加粗显示），使其在编译过程中包含新增加的C#文件：

```
<Project DefaultTargets="Build" xmlns=
  "http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>WpfAppAllXaml</RootNamespace>
    <AssemblyName>WpfAppAllXaml</AssemblyName>
    <OutputType>winexe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <ApplicationDefinition Include="MyApp.xaml" />
      <Compile Include = "MainWindow.xaml.cs" />
      <Compile Include = "MyApp.xaml.cs" />
      <Page Include="MainWindow.xaml" />
  </ItemGroup>
  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
  <Import Project="$(MSBuildBinPath)\Microsoft.WinFX.targets" />
</Project>
```

将构建脚本传递给msbuild.exe：

```
msbuild WpfAppAllXaml.csproj
```

你会发现再次生成了WpfAppAllXaml应用程序的可执行程序集（还记得吗，位于bin\Debug文件夹）。就开发而言，我们将外观（XAML）与编程逻辑（C#）进行了彻底分离。

这是WPF开发的推荐方法，在使用Visual Studio（或Expression Blend）创建WPF应用程序时，将总是使用这种代码隐藏模型。

源代码 WpfAppCodeFiles项目的源代码位于Chapter 27子目录下。

27.9 使用 Visual Studio 构建 WPF 应用程序

到现在为止，我们都使用普通的文本编辑器、命令行编译器和Kaxaml来创建示例。当然，这样做是为了更多关注WPF应用程序的核心语法，而不是图形化设计器的各种功能。然而，既然我们已经了解如何构建WPF应用程序，下面让我们研究一下Visual Studio是如何简化构建WPF程序工作的。

说明 我将在本章指出一些使用Visual Studio构建WPF应用程序的关键特征。下一章会在必要的时候演示这个IDE的其他部分。

27.9.1 WPF 项目模板

Visual Studio的New Project对话框定义了一组WPF相关的项目模板，可以在Visual C#根的Window节点下找到所有的项。此处，我们可以选择WPF Application、WPF User Control Library、WPF Custom Control Library和WPF Browser Application（如XBAP）。我们创建一个名为WpfTesterApp的WPF Application（如图27-15所示）。

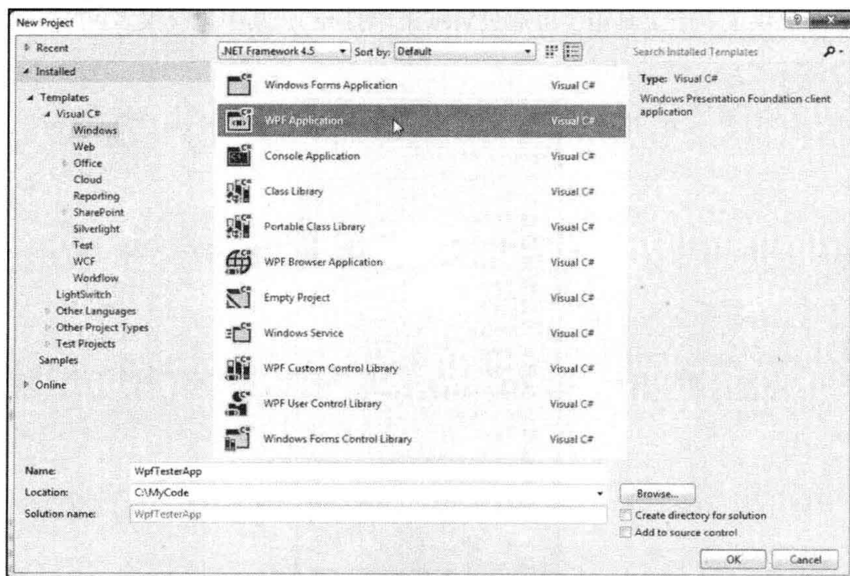


图 27-15 Visual Studio 的 WPF 项目模板位于 Windows 节点下

它除了添加对所有WPF程序集（PresentationCore.dll、PresentationFramework.dll、System.Xaml.dll和WindowsBase.dll）的引用之外，还为我们提供了使用代码文件和XAML表示的初始的Window和Application派生类型。图27-16所示显示了新的WPF项目的Solution Explorer窗口。

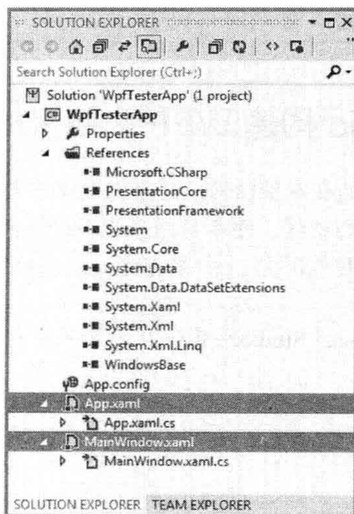


图27-16　WPF应用程序项目的初始文件

27.9.2　工具箱和XAML设计器/编辑器

Visual Studio提供了一个工具箱（可通过View菜单打开），其中包含很多WPF控件（如图27-17所示）。

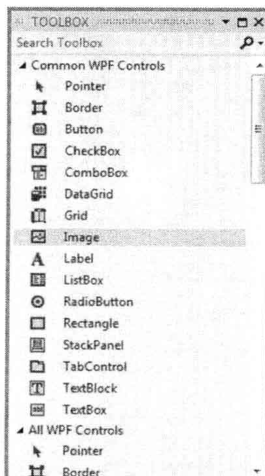


图27-17　工具箱所包含的WPF控件可以置于设计器表面上

使用标准的鼠标拖曳操作，可以将这些控件放置于窗口设计器上，或拖曳到设计器下方的XAML标记编辑器中。然后将自动生成XAML。用鼠标将一个Button和一个Calendar控件拖曳到设计器上。然后注意我们可以重新放置控件，或改变控件的尺寸（同时注意观察根据我们的编辑而生成的结果XAML）。

除了可以通过鼠标和工具箱构建UI，也可以在集成的XAML编辑器中手工输入这些标记。如图27-18所示，在输入标记时可得到智能感知支持。例如，在开放的`<Window>`元素中添加`Background`属性。

花几分钟时间直接在XAML编辑器中添加一些属性。我敢肯定还是在WPF设计器中添加属性更加轻松。

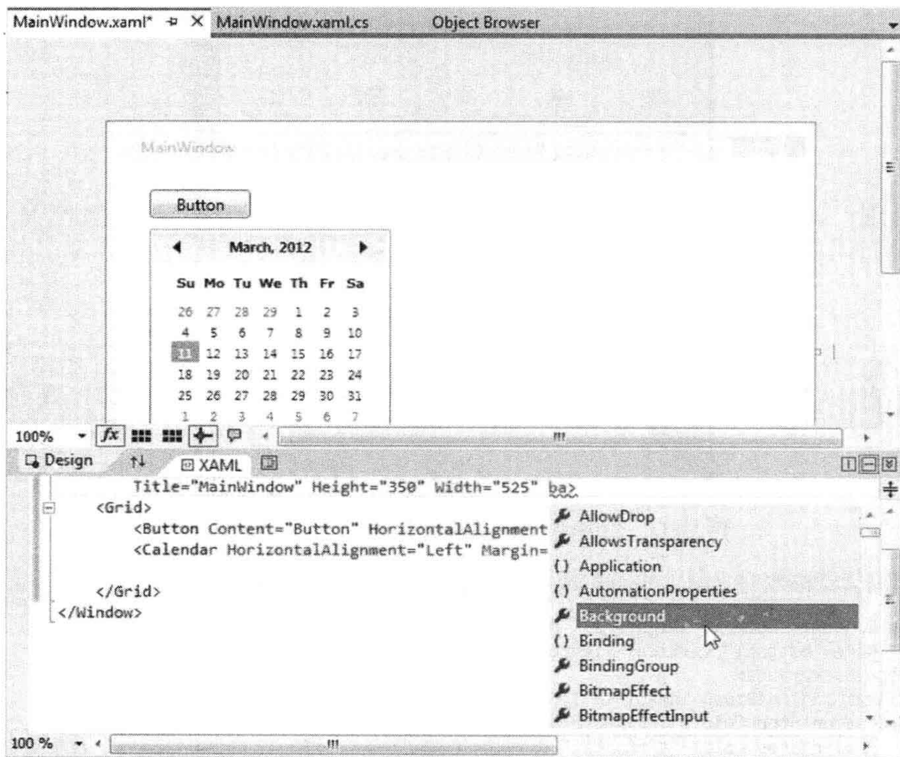


图27-18 WPF窗口设计器

27.9.3 使用Properties窗口设置属性

将控件放置到设计器上（或在编辑器中手工定义）之后，可以使用Properties窗口设置所选控件的属性值和事件处理程序。我们进行一个简单的测试，在设计器中选择Button控件。然后，使用Properties窗口中集成的画刷编辑器修改Button的背景色（如图27-19所示，我们将在第29章介绍WPF图像时详细介绍画刷编辑器）。

说明　Properties窗口的最上方有一个搜索文本框。可以在其中输入属性的名称以快速找到想要的项。

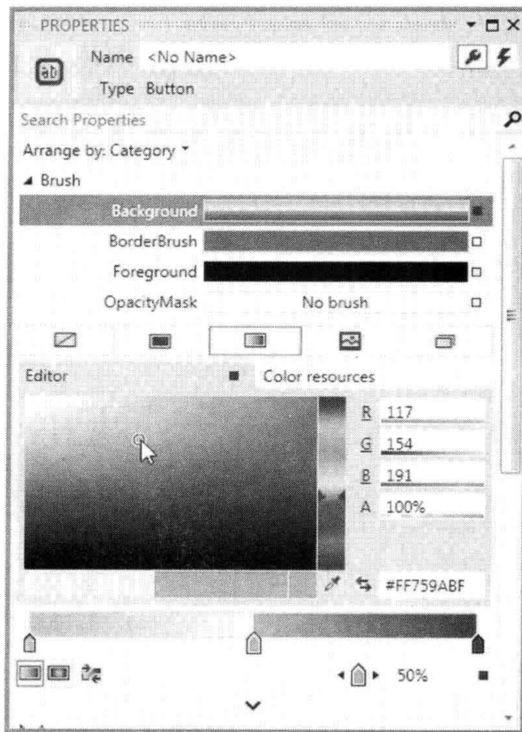


图27-19　Properties窗口可以用来配置WPF控件的UI

用画刷编辑器修改完之后，检查生成的标记，可能会如下所示：

```
<Button Content="Button" Height="23" HorizontalAlignment="Left" Margin="12,12,0,0"
        Name="button1" VerticalAlignment="Top" Width="75">
  <Button.Background>
    <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
      <GradientStop Color="#FF7488CE" Offset="0" />
      <GradientStop Color="#FFC11E1E" Offset="0.837" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

27.9.4　使用Properties窗口处理事件

如果你想处理给定控件的事件，也可以使用Properties窗口，但这时要点击Properties窗口右上方的Events按钮（找到闪电图标）。确保选中设计器中的按钮，单击Eventes选项卡，找到`Click`事件并双击右侧的条目。Visual Studio将自动构建如下形式的事件处理程序：

```
NameOfControl_NameOfEvent
```

由于没有修改按钮的名称，Properties窗口中显示生成的事件处理程序的名称为Button_Click_1（如图27-20所示）。

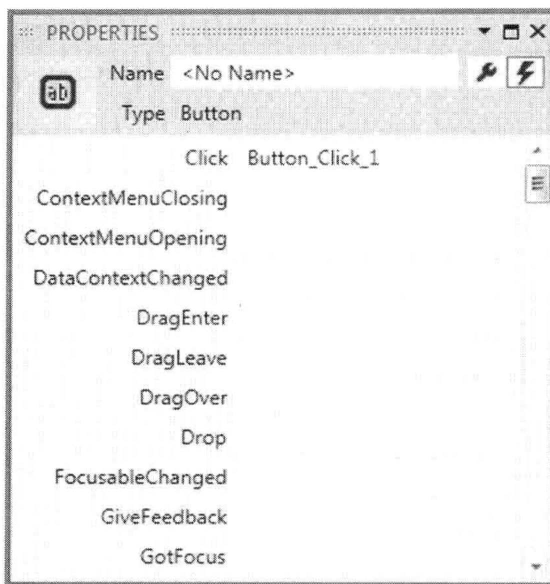


图27-20　使用Properties窗口处理事件

同样，Visual Studio还在窗口代码文件中生成了相应的C#事件处理程序。你可以在其中添加任何在点击按钮时执行的代码。比如，输入下面的代码语句：


```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  private void Button_Click_1(object sender, RoutedEventArgs e)
  {
    MessageBox.Show("You clicked the button!");
  }
}
```

27.9.5　在XAML编辑器中处理事件

你还可以在XAML编辑器中直接处理事件。例如，将光标放到<Window>元素内，输入MouseMove事件，然后输入等号，此时Visual Studio将显示所有代码文件中兼容的处理程序（如果代码文件存在的话），以及<New Event Handler>选项（如图27-21所示）。

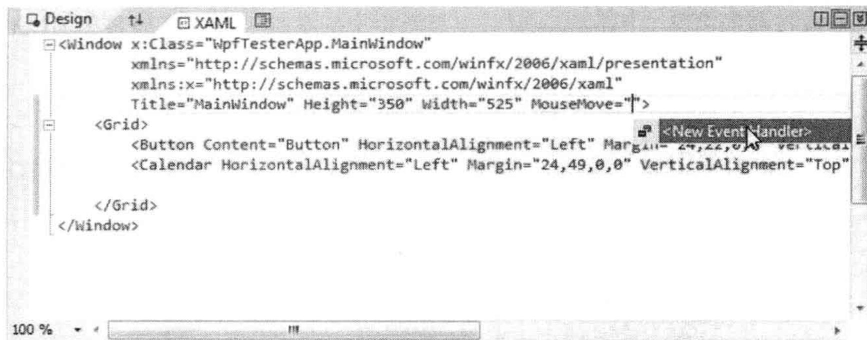


图27-21 使用XAML编辑器处理事件

双击`<New Event Handler>`，IDE将在C#代码文件中生成适当的处理程序。在MouseMove事件处理程序中输入下面的代码，然后运行程序查看最终结果：

```
private void Window_MouseMove (object sender, MouseEventArgs e)
{
  this.Title = e.GetPosition(this).ToString();
}
```

27.9.6 Document Outline窗口

在创建基于XAML的项目（WPF、Silverlight、Windows Phone 7或Windows 8应用程序）时，我们自然会用大量标记来表示UI。在处理更加复杂的XAML时，如果能够使标记可视，能快速选择Visual Studio设计器中的项进行编辑，将是非常有用的。

我们目前的标记都十分简单，只是在初始的`<Grid>`中定义了少许控件。不管这些，找到IDE的Documents Outline窗口，它默认位于IDE的左下角（如果找不到，可以通过View→ Other Windows菜单选项激活该窗口）。现在，确保你的XAML编辑器是IDE中当前的激活窗口（不是C#代码文件），我们可以看到Document Outline展示了内嵌的元素（如图27-22所示）。

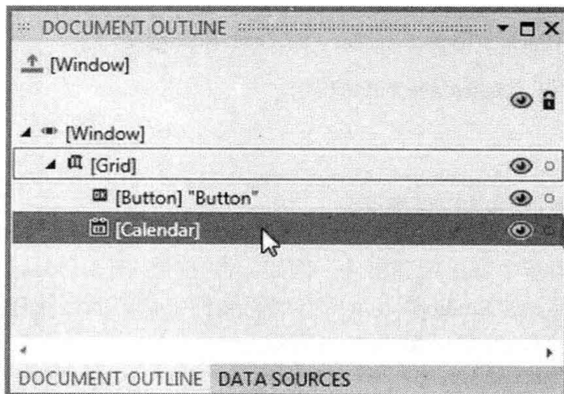


图27-22 通过Document Outline窗口使XAML可视化

这个工具还能临时在设计器上隐藏给定项，或锁定项以防止编辑。在下一章中，你将看到Document Outline窗口还提供了很多其他特性，如将选中项分组到新的布局管理器。

27.9.7 查看自动生成的代码文件

在构建本章最后一个示例之前，打开Solution Explorer窗口，单击Show All Files按钮（如图27-23所示）。注意目前显示的是BAML和*.g.cs文件（位于obj\Debug文件夹下）。虽然不推荐在自动生成的文件中手动添加代码，但本章前面的示例可以帮助我们理解如何处理XAML。

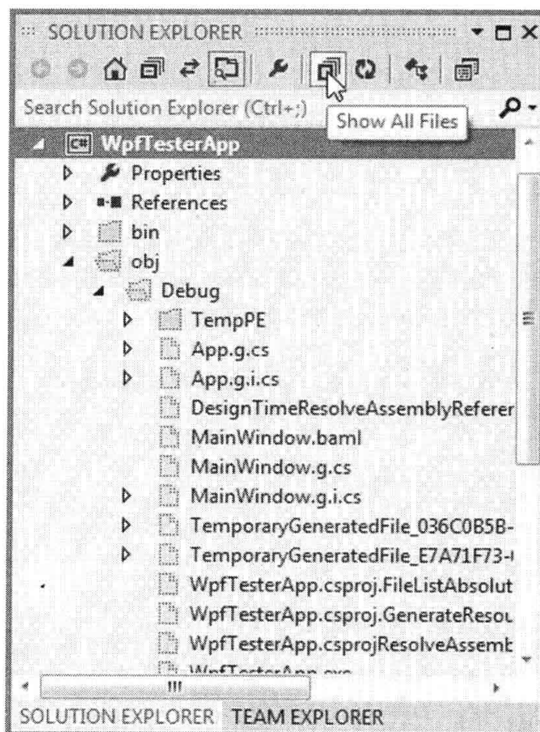


图27-23 使用Solution Explorer查看WPF项目的输出文件

27.10 使用 Visual Studio 构建自定义 XAML 编辑器

我们已经看到了如何在Visual Studio中使用基本工具设计WPF窗体，本章的最后一个示例将构建一个可以在运行时操作XAML的应用程序。关闭当前项目，创建一个全新的WPF Application，取名为MyXamlPad。该项目（在完成时）的功能将和Kaxaml类似，只是没那么炫。具体来说，该应用程序可以输入任何格式良好的标记，并且点击按钮即可将XAML动态呈现到新的Window对象中。

27

27.10.1　设计窗口的GUI

WPF API支持以编程方式加载、解析和保存XAML描述。对于很多情况，这么做是很有用的。例如，假设我们有5个不同的XAML文件，用于描述一个Window类型的外观。只要每个文件中所有控件（和必要的事件处理程序）名字都一样，我们就可以为窗口动态应用“皮肤”（可能根据传入应用程序的起始参数）。

运行时和XAML交互需要使用XamlReader和XamlWriter类型，它们都定义在System.Windows.Markup命名空间中。为了演示如何以编程方式从外部*.xaml文件生成Window对象，我们会创建一个WPF应用程序项目来模仿Kaxaml的基本功能。

说明　XamlReader和XamlWriter类提供了在运行时操纵XAML的基本功能。如果想完全控制XAML对象模型，可以研究一下System.Xaml.dll程序集。

虽然我们的应用程序肯定不会和Kaxaml一样功能完善，但是它提供了输入有效XAML标记、查看结果并且保存XAML到外部文件的能力。首先更新<Window>中初始的XAML定义，如下（建议此时手工输入这些XAML，而像前面那样用IDE生成事件处理程序）：

```
<Window x:Class="MyXamlPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="My Custom XAML Editor"
  Height="338" Width="1041"
  Loaded="Window_Loaded" Closed="Window_Closed"
  WindowStartupLocation="CenterScreen">

  <!-- 你将会使用DockPanel，而不是Grid -->
  <DockPanel LastChildFill="True" >

    <!-- 这个按钮会启动具有已定义的XAML的窗口 -->
    <Button DockPanel.Dock="Top" Name = "btnViewXaml" Width="100" Height="40"
            Content ="View Xaml" Click="btnViewXaml_Click" />

    <!-- 它是用于输入的区域 -->
    <TextBox AcceptsReturn ="True" Name ="txtXamlData"
             FontSize ="14" Background="Black" Foreground="Yellow"
             BorderBrush ="Blue" VerticalScrollBarVisibility="Auto"
             AcceptsTab="True"/>
  </DockPanel>
</Window>
```

说明　下一章会深入使用控件和面板的细节，因此不要对控件的声明感到烦躁。

首先，注意我们把初始的<Grid>替换为<DockPanel>布局管理器了，它包含一个Button（叫做btnViewXaml）和一个TextBox（叫做txtXamlData），Button类型的Click事件也被处理了。

还注意Window自身的Loaded和Closed事件在<Window>开始元素中被处理了（同样，应该像本章前几

节那样使用IDE来生成事件处理程序）。如果你使用设计器来处理事件的话，就会发现在MainWindow.xaml.cs 文件中具有如下的代码：

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  private void btnViewXaml_Click(object sender, RoutedEventArgs e)
  {
  }

  private void Window_Closed(object sender, EventArgs e)
  {
  }

  private void Window_Loaded(object sender, RoutedEventArgs e)
  {
  }
}
```

在继续之前，请一定要为MainWindow.xaml.cs文件导入如下的命名空间：

```
using System.IO;
using System.Windows.Markup;
```

27.10.2 实现Loaded事件

主窗口的Loaded事件负责检测在包含应用程序的文件夹中是否有一个名为YourXaml.xaml的文件。如果文件存在的话，就会读取其中的数据并放在主窗口的TextBox中。如果没有的话，就会使用空窗口默认的XAML描述来填充TextBox（这个描述和初始的窗口定义一样，只是使用<StackPanel>而不是<Grid>来隐式设置Window的Content属性）。

说明 我们构建的表示在编辑器中显示的初始标记的字符串很难输入，因为需要为引用内容做转义，请小心输入。

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  //应用程序主窗口加载的时候，
  //把一些基本的XAML文本放到文本框内
  if (File.Exists("YourXaml.xaml"))
  {
    txtXamlData.Text = File.ReadAllText("YourXaml.xaml");
  }
  else
  {
    txtXamlData.Text =
    "<Window xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\"\n"
    +"xmlns:x=\"http://schemas.microsoft.com/winfx/2006/xaml\"\n"
    +"Height =\"400\" Width =\"500\" WindowStartupLocation=\"CenterScreen\">\n"
    +"<StackPanel>\n"
```

```
    +"</StackPanel>\n"
    +"</Window>";
  }
}
```

使用这个方式，应用程序就可以加载前面会话中输入的XAML，或必要的时候提供默认的标记片段。至此，你应该可以运行程序并且在TextBox类型中找到如图27-24所示的显示。

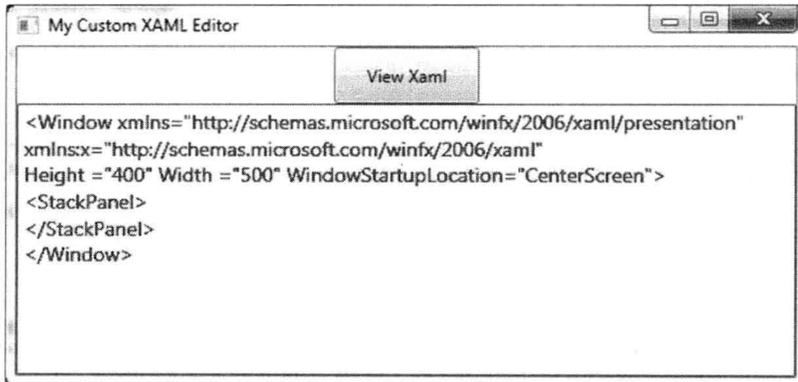


图27-24　MyXamlPad.exe首次运行

27.10.3　实现按钮的Click事件

如果你单击Button类型，就会先把TextBox中当前的数据保存到YourXaml.xaml文件中。至此，你就可以通过File.Open()读取持久化的数据来获取FileStream。这是必需的，因为XamlReader.Load()方法需要一个Stream派生类型（而不是简单的System.String）来表示要解析的XAML。

在加载了你希望构建的<Window>的XAML描述之后，根据内存中的XAML创建System.Windows.Window的实例，并且以模式对话框显示Window，如下所示：

```
private void btnViewXaml_Click(object sender, RoutedEventArgs e)
{
  // 把文本框中的数据写入本地的*.xaml文件中
  File.WriteAllText("YourXaml.xaml", txtXamlData.Text);

  // 这个Window XAML会被动态解析
  Window myWindow = null;
  // 打开本地*.xaml文件
  try
  {
    using (Stream sr = File.Open("YourXaml.xaml", FileMode.Open))
    {
      // 把XAML连接到Window对象
      myWindow = (Window)XamlReader.Load(sr);

      // 将窗口显示为对话框并清除掉
      myWindow.ShowDialog();
      myWindow.Close();
      myWindow = null;
```

```
        }
      }
      catch (Exception ex)
      {
        MessageBox.Show(ex.Message);
      }
    }
```

注意，我们使用try/catch块来包装我们很多的逻辑。这样，如果YourXaml.xaml包含了非法格式的标记，就可以在结果对话框中看到错误。例如，运行程序，故意拼错<StackPanel>，如多写了一个P。单击按钮，将看到类似图27-25所示的错误。

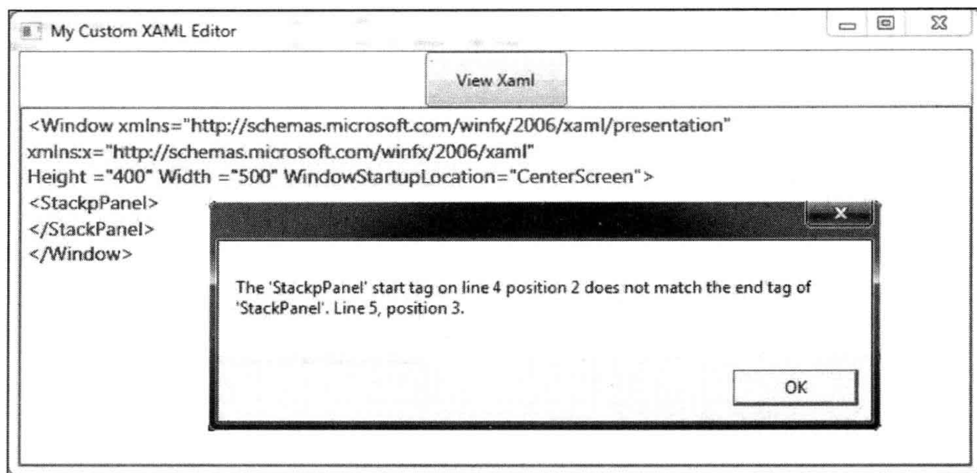


图27-25 捕获标记错误

27

27.10.4 实现 Closed 事件

最后，Window类型的Closed事件会确保TextBox中最新的数据持久化到YourXaml.xaml文件中：

```
private void Window_Closed(object sender, EventArgs e)
{
  // 把文本框中的数据写到本地*.xaml文件中
  File.WriteAllText("YourXaml.xaml", txtXamlData.Text);
  Application.Current.Shutdown();
}
```

27.10.5 测试应用程序

现在启动程序并且在文本区域中输入一些XAML。要知道（和Kaxaml一样），这个程序不允许我们指定任何代码生成相关的XAML特性（如Class或事件处理程序）。作为第一个测试，在<StackPanel>区域内输入下面的XAML：

```
<Button Height = "100" Width = "100" Content = "Click Me!">
  <Button.Background>
    <LinearGradientBrush StartPoint = "0,0" EndPoint = "1,1">
```

```
      <GradientStop Color = "Blue" Offset = "0" />
      <GradientStop Color = "Yellow" Offset = "0.25" />
      <GradientStop Color = "Green" Offset = "0.75" />
      <GradientStop Color = "Pink" Offset = "0.50" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

在单击了按钮之后，我们就会看到一个窗口呈现了我们的XAML定义。（也许会看到一个消息框有个错误解析，它在检查输入！）图27-26显示了可能的输出。

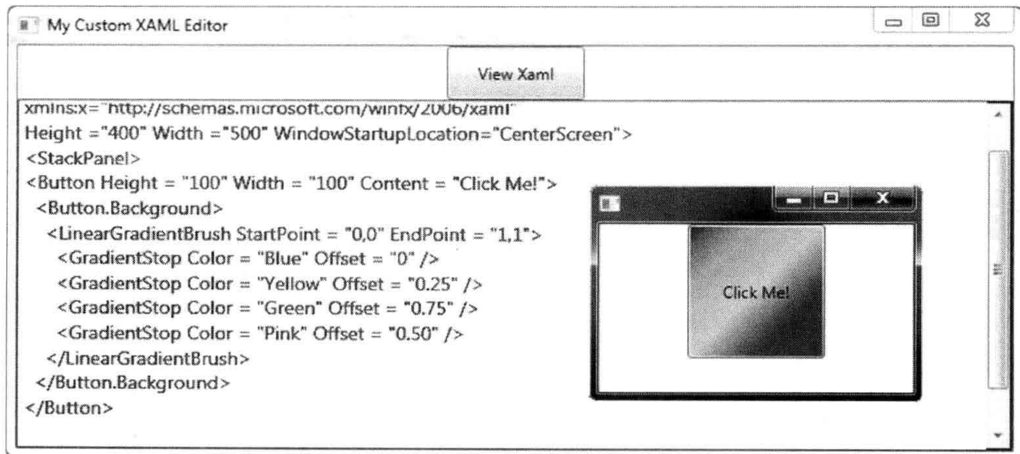


图27-26　MyXamlPad.exe实战

现在，在<Button>定义后面直接输入下面的XAML标记：

```
<Label Content = "Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent = "Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "FontSize">
            <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
              RepeatBehavior = "Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

这一标记很好地阐明了XAML的真正强大之处。测试该标记，你会发现创建了一个简单的动画序列。动画服务（和图像呈现）将在后续章节详细介绍。不过你可以调整这里的XAML来查看最终结果。

27.10.6　探索WPF 文档

在本章最后，我想指出的是，.NET 4.5 Framework SDK文档有一个完整的章节专门介绍WPF主题。在学习这个API和阅读本书其他WPF相关章节的时候，如果及时并经常参考帮助系统的话，对你来说

将是莫大的帮助。你在那里可以看到大量XAML示例、详细的教程（主题广泛，从3D图像编程到复杂的数据绑定操作）。

WPF文档位于.NET Framework 4.5→.NET Framework Development Guide→Developing Client Applications下（如图27-27所示）。

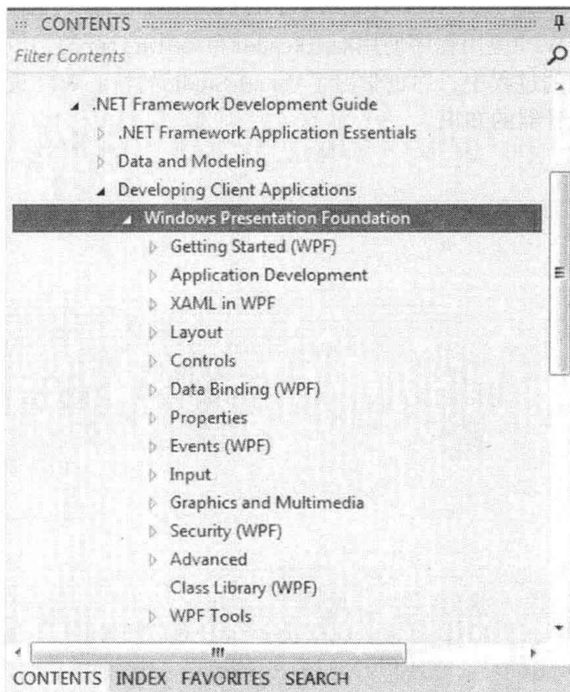


图27-27 NET 4.5 Framework SDK文档提供了多方位的WPF帮助

在查看帮助系统时，你会发现大量的XAML示例，可以直接复制到剪贴板并粘贴到自定义的XAML编辑器中。但是，在测试前要确保根元素从`<Page>`改成`<Window>`（我们的应用程序不是显示`Page`对象，而是`Window`对象）。在开始下一章之前，你可以花些时间深入感兴趣的话题，并在自定义的工具里测试其他标记。

源代码 MyXamlPad项目的源代码位于Chapter 27子目录下。

27.11 小结

WPF是.NET 3.0发布后引入的用户界面工具包。WPF的主要目的是将许多之前无关的桌面技术（2D图形、3D图形、窗口和控件开发等）整合到一个统一的编程模型中。此外，WPF程序一般使用XAML，它可以通过标记来声明WPF元素的外观。

27

在本章中我们已经看到了，XAML可以使用声明语法描述.NET对象树。在本章研究XAML的过程中，我们研究了包括属性元素语法、附加属性以及类型转换器和XAML标记扩展在内的新语法。

尽管对于任何产品级的WPF应用程序来说，XAML都是不可或缺的一部分，但本章的第一个示例演示的是如何仅使用C#代码来构建WPF程序。然后演示了如何仅使用XAML（我们并不推荐这么做，但这是一次有用的学习训练）构建WPF程序。最后，我们学习了使用“代码文件”将外观和功能分离。

本章构建的最后一个WPF应用程序使用XamlReader和XamlWriter类，以编程方式与XAML定义进行交互。在介绍这些示例的过程中，你也学习了Visual Studio的核心WPF设计器。在后面的章节，你还将学习更多关于WPF设计器的知识。

第 28 章

使用WPF控件编程

28

第27章讲解了WPF编程模型的基础，包括Window和Application类的介绍、XAML的语法和代码文件的用法。第27章还介绍了使用Visual Studio设计器构建WPF应用程序的过程。本章将使用一些新的控件和布局管理器，并学习Visual Studio中WPF设计器的其他特性。

本章还将介绍一些与WPF控件相关的重要话题，如数据绑定编程模型和控件命令的用法。你将学习如何使用Ink和Documents API，来获取手写笔（或鼠标）输入，并使用XML Paper Specification（XPS）构建富文本文档。

说明 本书上一版使用了微软的Expression Blend产品来简化使用WPF API构建GUI的过程。不过，最新版的Visual Studio提供了足够的功能来构建本书所设计的WPF UI。如果要学习使用Expression Blend的细节，可以参阅笔者的另一本书*Pro Expression Blend 4*（2011，Apress）。

28.1 WPF 核心控件概述

不管过去使用过哪种GUI工具包（VB 6.0、MFC、Java AWT/Swing、Windows Forms、Mac OS X[Cocoa]和GTK+/GTK#等），除非你对创建图形用户界面非常陌生，否则主要WPF控件的普通用途对你来说不是什么问题。表28-1列出的WPF核心控件看起来非常熟悉。

表28-1 WPF的核心控件

WPF控件类别	成员示例	说 明
核心用户输入控件	Button、RadioButton、ComboBox、CheckBox、Calender、DatePicker、Expander、DataGrid、ListBox、ListView、ToggleButton、TreeView、ContextMenu、ScrollBar、Slider、TabControl、TextBlock、TextBox、RepeatButton、RichTextBox、Label	WPF提供了完整的控件族，用于创建用户界面的核心
窗口修饰控件	Menu、ToolBar、StatusBar、ToolTip、ProgressBar	这些UI元素用于装饰Window对象框架，它们包含输入工具（如Menu）和用户信息元素（StatusBar、ToolTip等）
媒体控件	Image、MediaElement、SoundPlayerAction	这些控件支持音频/视频的重放和图像的显示

（续）

WPF控件类别	成员示例	说　明
布局控件	Border、Canvas、DockPanel、Grid、GridView、GridSplitter、GroupBox、Panel、TabControl、StackPanel、Viewbox、WrapPanel	为了便于布局管理，WPF提供了许多用来分类和组织其他控件的控件

28.1.1　WPF Ink控件

除了表28-1中列出的常用WPF控件外，WPF还定义了其他一些使用了数字Ink API的控件。这对平板PC的开发是十分有用的，因为它可以捕获手写笔输入。但这并不是说标准的桌面应用不能使用Ink API，因为同样的控件也可以使用鼠标捕获输入。

PresentationCore.dll中的System.Windows.Ink命名空间包含多个Ink API支持的类型（如Stroke和strokecollection）。但是，大部分Ink API控件（如InkCanvas和InkPresenter）及常用的WPF控件都被打包在PresentationFramework.dll程序集的System.Windows.Controls命名空间下。本章稍后将介绍Ink API。

28.1.2　WPF Document控件

WPF还为高级文档处理提供了控件，使我们构建的应用程序拥有Adobe PDF风格。使用System.Widows.Documents命名空间（同样位于PresentationFramework.dll程序集）中的类型，可以创建用来打印的文档，它们支持缩放、搜索、用户批注（便签）和其他富文本服务。

然而在后台，文档控件并没有使用Adobe PDF API，而是用XPS API。对于最终用户来说，显示起来没有任何区别，因为PDF文档和XPS文档有着几乎完全相同的外观。事实上，很多免费工具都可以在运行时对这两种文件进行相互转换。在接下来的示例中，我们会使用该文档控件的一部分功能。

28.1.3　WPF公共对话框

WPF还提供了一些公共对话框，如OpenFileDialog和SaveFileDiaiog，这些对话框都定义在PresentationFramework.dll程序集的Microsoft.Win32命名空间下。使用它们都要创建一个对象，然后调用ShowDialog()方法，如下所示：

```
using Microsoft.Win32;

namespace WpfControls
{
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }

    private void btnShowDlg_Click(object sender, RoutedEventArgs e)
    {
      // 显示文件保存对话框
      SaveFileDialog saveDlg = new SaveFileDialog();
      saveDlg.ShowDialog();
```

```
        }
    }
}
```

如你所料，这些类的成员用来建立文件过滤器和目录路径，并访问用户选择的文件。后面的示例将使用这些文件对话框，你还将学习如何构建自定义对话框以收集用户输入。

28.1.4　文档中的细节

不管你怎么想，本章的目的并不是要介绍每个WPF控件中的每个成员，而是要概述核心控件，并强调基本的编程模型（依赖属性、路由事件等命令等）和大部分WPF控件都有的核心服务。

为了加深对给定控件的功能的理解，请查阅.NET Framework 4.5 SDK文档，特别是帮助系统的控件库一节，它位于Windows Presentation Foundation→Controls（如图28-1所示）。

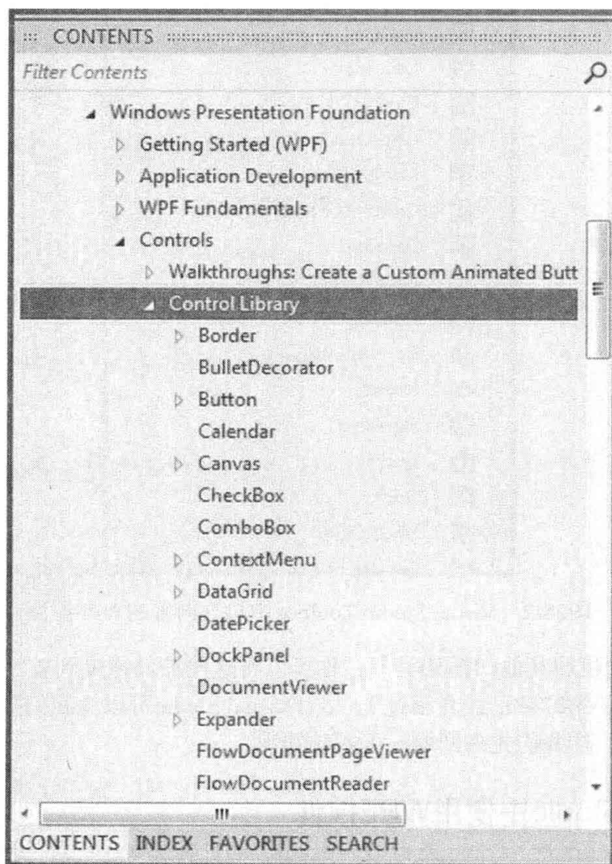


图28-1　每个WPF控件完整的介绍可以通过按F1键取得

在这里，可以查找到每个控件的完整介绍、各种代码示例（XAML和C#）、关于控件继承链的信息、实现的接口和应用的特性。一定要花点时间浏览本章所介绍的这些控件的完整细节。

28.2 Visual Studio WPF 设计器

这些标准的WPF控件大多位于PresentationFramework.dll程序集的System.Windows.Controls命名空间下。使用Visual Studio构建WPF应用程序时，如果激活一个WPF设计器，就可以在Toolbox中看到其中大多数常见控件（如图28-2所示）。

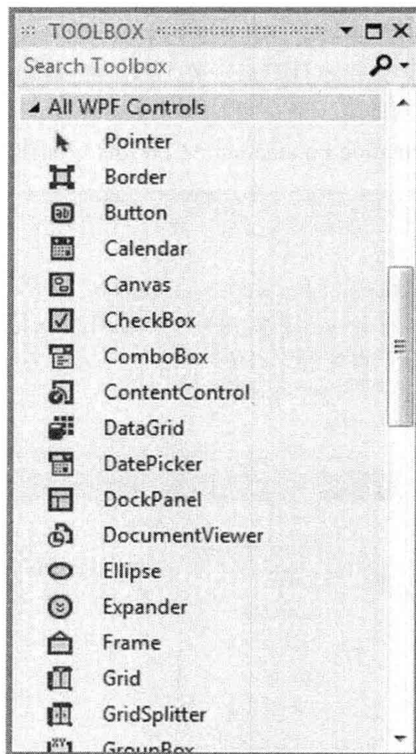


图28-2 Visual Studio Toolbox中包含内置的WPF控件

和用Visual Studio创建的其他UI框架类似，你可以将这些控件拖曳到WPF窗口设计器上，并使用Properties窗口进行配置（第27章已经介绍过）。尽管Visual Studio可以为我们生成大量XAML，但手工编辑标记也是很常见的。让我们先来回顾一下基础知识。

28.2.1 在Visual Studio中使用WPF控件

第27章介绍过，当在Visual Studio的设计器中放置WPF控件时，可以通过Properties窗口设置x:Name属性，因为这样就可以在C#代码文件中访问该对象。你可能还记得我们可以使用Properties窗口中的Events选项卡，为选中的控件生成事件处理程序。因此，你可以使用Visual Studio为一个简单的Button控件生成如下标记：

```
<Button x:Xame="btnMyButton" Content="Click Me!" Height="23" Width="140"
        Click="btnMyButton_Click" />
```

你可以将Button的Content属性设置为简单的字符串“Click Me!”。但有了WPF控件内容模型，你还可以让Button包含如下复杂的内容：

```
<Button x:Name="btnMyButton" Height="121" Width="156" Click="btnMyButton_Click">
  <Button.Content>
    <StackPanel Height="95" Width="128" Orientation="Vertical">
      <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
      <Label Width="59" FontSize="20" Content="Click!" Height="36" />
    </StackPanel>
  </Button.Content>
</Button>
```

你也许还记得，ContentControl派生类的第一个子元素即为隐含的内容。因此，在指定复杂内容时，没有必要显式定义<Button.Content>作用域。你可以简单地编写如下标记：

```
<Button x:Name="btnMyButton" Height="121" Width="156" Click="btnMyButton_Click">
  <StackPanel Height="95" Width="128" Orientation="Vertical">
    <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
    <Label Width="59" FontSize="20" Content="Click!" Height="36" />
  </StackPanel>
</Button>
```

在这两种方法中，我们都将按钮的Content属性设置为相关项的<StackPanel>。你还可以使用Visual Studio的设计器来指定这种复杂内容。在为内容控件定义好布局管理器后，在设计器中选中它，就可以将其作为内部控件的拖曳目标。然后，就可以在Properties窗口中编辑它们了。如果使用Properties窗口来处理Button控件的Click事件（如前面的XAML声明所示），IDE将生成空的事件处理程序，你可以添加自定义代码。例如：

```
private void btnMyButton_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("You clicked the button!");
}
```

28

28.2.2 使用Document Outline编辑器

你还应该注意Visual Studio的Document Outline窗口（可以通过View→Other Windows菜单打开），它在设计包含复杂内容的WPF控件时是十分有用的。注意图28-3中显示了正在构建的Window的XAML逻辑树。单击某个节点，将在设计器中自动选中，并可以编辑。

当前版本Visual Studio中的Document Outline编辑器包含了其他一些非常有用的特性。注意在节点右边有一个看上去像眼球的图标。点击该按钮，可以在设计器中显示或隐藏某一项，这在你想集中注意力编辑特定的片段时是非常有用的（注意，这只是在设计器表明隐藏指定的项，不会在运行时隐藏该项）。

“眼球图标”右边的按钮可以在设计器中“锁定”某一项。你应该能猜到它的功能：这可以有效地防止你（或你的同事）不小心改变了给定项的XAML。实际上，锁定项意味着在设计时将其置为只读（在运行时仍然可以改变其状态）。

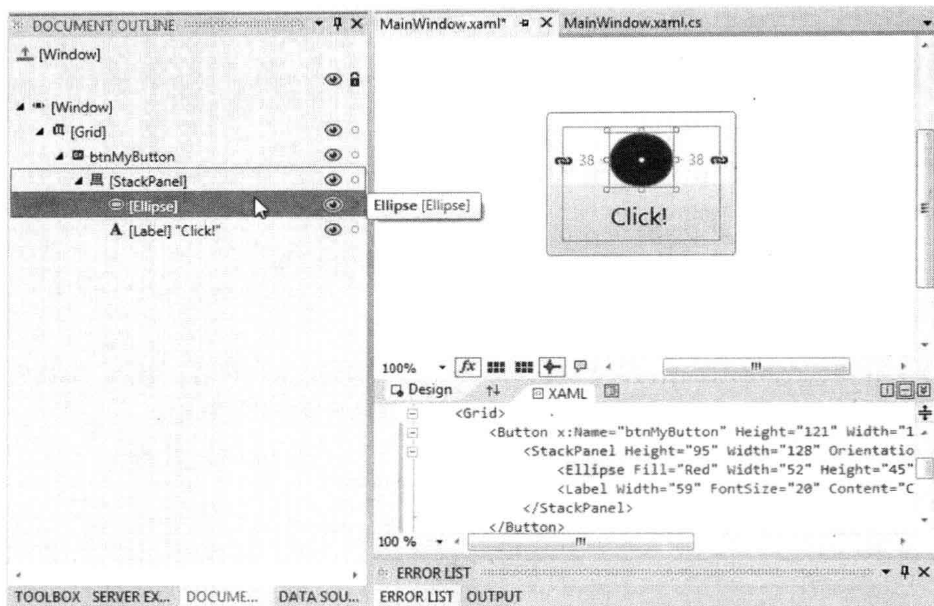


图28-3　Visual Studio中的Document Outline窗口可以帮助你在复杂的内容中进行导航

28.3　使用面板控制内容布局

WPF应用程序总是包含很多UI元素（如用户输入控件、图形内容、菜单系统和状态栏），它们需要在不同的窗口内进行良好的组织。在放置了UI元素之后，你需要确保它们在用户调整窗口或窗口的一部分（如拆分窗口）大小的时候，其表现形式也能如你所期。为了让WPF控件能够保持在宿主窗口中的位置，你可以使用面板类型（作为布局管理器）。

默认情况下，使用Visual Studio创建的WPF窗体使用<Grid>类型的布局管理器（详情稍后介绍）。不过我们现在假设窗体没有声明布局管理器。如下所示：

```
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
...
  Title="Fun with Panels!" Height="285" Width="325">

</Window>
```

如果在不使用面板的窗口中直接声明控件，那么控件将被放置在容器的正中央。考虑下面这个简单的窗口声明，它包含一个Button控件。不管如何调整窗口大小，UI控件始终位于客户端区域的中央。Button的大小取决于设置的Height和Width属性：

```
<!-- 该按钮始终位于窗口中央 -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
...
  Title="Fun with Panels!" Height="285" Width="325">

  <Button x:Name="btnOK" Height = "100"
          Width="80" Content="OK"/>
</Window>
```

你也许还记得，如果试图直接在`<Window>`作用域内放置多个元素，将得到标记和编译时错误。原因是窗口（或任何`ContentControl`的子类）的`Content`属性只能设置一个对象。因此，下面的XAML会产生标记和运行时错误：

```
<!-- 错误! Content属性被显式设置了多次-->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">

  <!-- 错误! <Window>有两个直接子元素-->
  <Label x:Name="lblInstructions" Width="328" Height="27"
         FontSize="15" Content="Enter Information"/>
  <Button x:Name="btnOK" Height = "100" Width="80" Content="OK"/>
</Window>
```

只能容纳一个控件的窗口显然没有任何用途。如果窗口需要包含多个元素，那么这些元素必须被安排在多个面板中。面板可以容纳所有表示窗口的UI元素，而面板本身将作为唯一的对象，赋给窗口的`Content`属性。

`System.Windows.Controls`命名空间提供了很多面板，它们控制着如何维护子元素。如果最终用户调整窗口大小，或控件要保留在设计时的位置，或控件从左到右水平回流或从上到下垂直回流等，你都可以使用面板建立控件的行为。

为了提供更加灵活的控制，你还可以在面板中使用其他面板（如`DockPanel`中包含`StackPanel`）。表28-2列出了一些常用WPF面板控件的作用。

表28-2 核心WPF面板控件

面板控件	作　用
`Canvas`	提供了一个典型的内容布置模式。其中的项将保留在设计时所放置的确切位置
`DockPanel`	将内容锁定在面板的某一侧（`Top`、`Bottom`、`Left`、`Right`）
`Grid`	将内容安排在一系列单元格中，像表格一样进行维护
`StackPanel`	将内容按水平或垂直方式进行堆放，具体方式由`Orientation`属性表示
`WrapPanel`	从左向右按顺序放置内容，在框的边缘处将内容断开至下一行。后续排序按照从上到下或从右到左的顺序进行，具体取决于`Orientation`属性的值

在下面的几节中，我们会将一些预定义的XAML数据复制到第27章创建的MyXamlPad.exe应用程序中（如果愿意，你也可以在kaxaml.exe中加载这些数据），以此来学习如何使用这些常用的面板类型。你可以在代码下载资源的Chapter 28目录下的PanelMarkup子目录下找到这些XAML文件（如图28-4所示）。

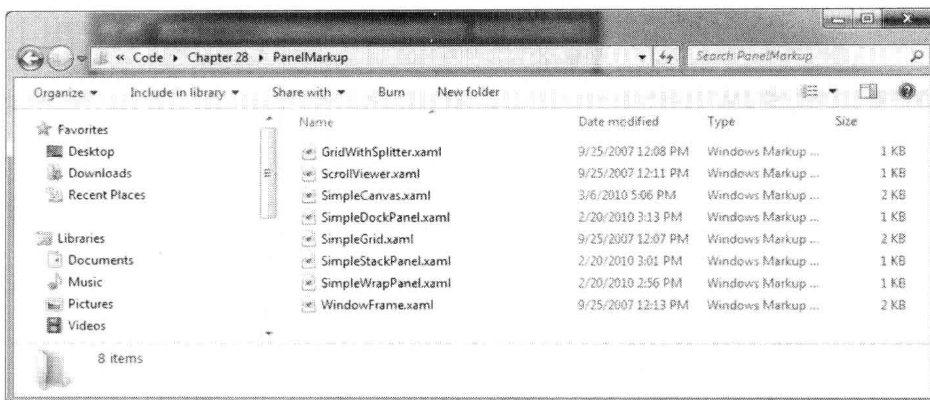


图28-4　为了测试不同的布局，我们将在MyXamlPad.exe应用程序中加载提供的XAML数据

28.3.1　在Canvas面板中放置内容

在使用Canvas面板时你可能会觉得很方便，因为它允许UI内容使用绝对位置。如果用户调整窗口的尺寸，使其小于Canvas面板所需的布局，那么一部分内容将不可见，除非容器的尺寸等于或大于Canvas的区域。

要在Canvas中添加内容，先要在<Canvas>和</Canvas>标记作用域内定义所需的控件。然后，指定每个控件的左上角，这时要使用Canvas.Top和Canvas.Left属性。控件的右下角可以通过设置其Height和Width属性来间接指定，或直接使用Canvas.Right和Canvas.Bottom属性指定。

我们在文本编辑器中打开提供的SimpleCanvas.xaml文件，将其内容复制到MyXamlPad.exe（或kaxaml.exe）中。你可以看到如下的Canvas定义：

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
...
  Title="Fun with Panels!" Height="285" Width="325">
  <Canvas Background="LightSteelBlue">
    <Button x:Name="btnOK" Canvas.Left="212" Canvas.Top="203"
            Width="80" Content="OK"/>
    <Label x:Name="lblInstructions" Canvas.Left="17" Canvas.Top="14"
           Width="328" Height="27" FontSize="15"
           Content="Enter Car Information"/>
    <Label x:Name="lblMake" Canvas.Left="17" Canvas.Top="60"
           Content="Make"/>
    <TextBox x:Name="txtMake" Canvas.Left="94" Canvas.Top="60"
             Width="193" Height="25"/>
    <Label x:Name="lblColor" Canvas.Left="17" Canvas.Top="109"
             Content="Color"/>
    <TextBox x:Name="txtColor" Canvas.Left="94" Canvas.Top="107"
             Width="193" Height="25"/>
    <Label x:Name="lblPetName" Canvas.Left="17" Canvas.Top="155"
           Content="Pet Name"/>
    <TextBox x:Name="txtPetName" Canvas.Left="94" Canvas.Top="153"
```

```
            Width="193" Height="25"/>
  </Canvas>
</Window>
```

单击View Xaml按钮，可以在屏幕上看到如下所示的窗口（如图28-5所示）。



图28-5 在Canvas布局管理器中可以使用内容的绝对位置

注意，用来计算内容位置的，不是在Canvas中声明内容的顺序，而是控件的大小以及Canvas.Top、Canvas.Bottom、Canvas.Left和Canvas.Right属性。

说明 如果Canvas中的子元素没有使用附加属性语法（如Canvas.Left和Canvas.Top）定义具体的位置，那么它们将自动附加到Canvas的左上角。

Canvas类型看上去应该为布置内容的首选方式（因为它很简单），但这种方法却受到一些限制。首先，在使用样式或模板时，Canvas中的项不能自动调整大小（如字体大小不会受到影响）。其次，当用户将窗口缩小时，Canvas可能无法显示某些元素。

最适合在Canvas中放置的大概是图形内容。例如，如果使用XAML构建自定义图像，你当然希望线条、形状和文本都保留在固定的位置，而不希望在用户改变窗口大小时看到它们动态地改变位置。在第29章介绍WPF图形呈现服务时，我们将再次接触Canvas。

28.3.2 在WrapPanel面板中放置内容

WrapPanel中的内容将随窗口大小的变化而变化。在WrapPanel中放置元素时，不用像Canvas那样指定上、下、左、右的距离。但你仍然可以为每个子元素定义Height和Width值来控制其在容器中的整个尺寸。

由于WrapPanel中的内容没有指定到面板某一侧的距离，所以声明元素的顺序就显得十分重要（内容将按顺序逐个呈现）。加载SimpleWrapPanel.xaml文件中的XAML数据，可以看到如下标记（定义在<Window>之中）：

```
<WrapPanel Background="LightSteelBlue">
  <Label x:Name="lblInstruction" Width="328"
         Height="27" FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox x:Name="txtMake" Width="193" Height="25"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor" Width="193" Height="25"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName" Width="193" Height="25"/>
  <Button x:Name="btnOK" Width="80" Content="OK"/>
</WrapPanel>
```

加载这些标记，在调整窗口宽度时，内容将会被打乱，因为它们在窗口中是从左至右依次排列的（如图28-6所示）。

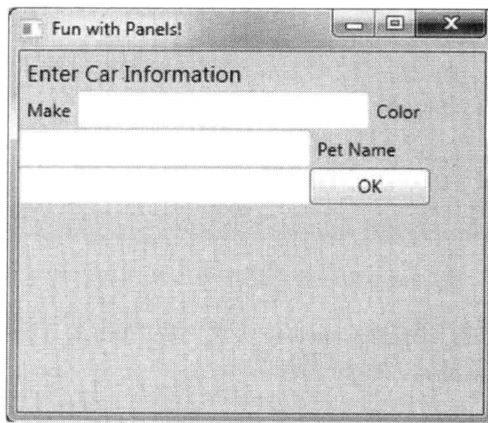


图28-6　WrapPanel中内容的行为更像是传统的HTML页面

默认情况下，WrapPanel中的内容从左向右排列。但如果将Orientation属性的值改为Vertical，内容则会从上到下排列：

```
<WrapPanel Background="LightSteelBlue" Orientation ="Vertical">
```

在声明WrapPanel时（对于其他面板类型也是一样），可以指定ItemWidth和ItemHeight值，来控制每个子项的默认尺寸。如果某个子元素没有提供自己的Height或Width值，将按面板中指定的尺寸来放置。考虑如下的标记：

```
<WrapPanel Background="LightSteelBlue" ItemWidth ="200" ItemHeight ="30">
  <Label x:Name="lblInstruction"
         FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox x:Name="txtMake"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName"/>
  <Button x:Name="btnOK" Width ="80" Content="OK"/>
</WrapPanel>
```

这些代码呈现时的输出结果如图28-7所示（注意Button控件的尺寸和位置，它指定了一个单独的Width值）。

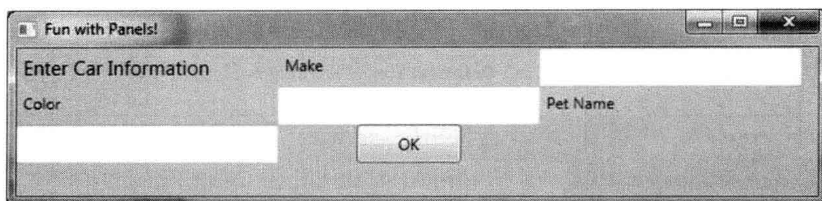


图28-7 WrapPanel可为给定项建立宽度和高度

看完图28-7后，你可能会发现，WrapPanel并不是在窗口中直接放置内容的最佳选择，因为在用户调整窗口大小时，其内部元素将变得混乱不堪。在大多数情况下，WrapPanel都将作为另一个面板类型的子元素，在调整窗口大小时，WrapPanel中的内容将会折行显示（如Toolbar控件）。

28.3.3 在StackPanel面板中放置内容

与WrapPanel类似，StackPanel控件也会根据Orientation属性的值，将内容水平或垂直（默认）地放置在一行中。但不同的是，在用户调整窗口时，StackPanel不会将内容折行，其中的项会根据面板设置的方向简单地拉伸，以适应StackPanel的大小。例如，SimpleStackPanel.xaml文件包含的标记如下，其输出结果如图28-8所示。

```
<StackPanel Background="LightSteelBlue">
  <Label x:Name="lblInstruction"
         FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" Content="Make"/>
  <TextBox Name="txtMake"/>
  <Label x:Name="lblColor" Content="Color"/>
  <TextBox x:Name="txtColor"/>
  <Label x:Name="lblPetName" Content="Pet Name"/>
  <TextBox x:Name="txtPetName"/>
  <Button x:Name="btnOK" Width ="80" Content="OK"/>
</StackPanel>
```

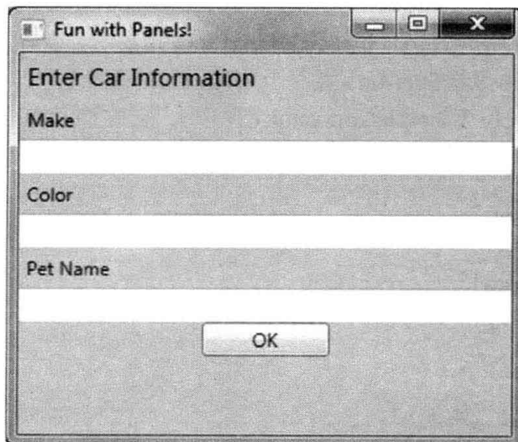


图28-8 垂直堆放的内容

如果将Orientation属性设置为Horizontal，呈现的输出结果将如图28-9所示。

```
<StackPanel Background="LightSteelBlue" Orientation ="Horizontal">
```

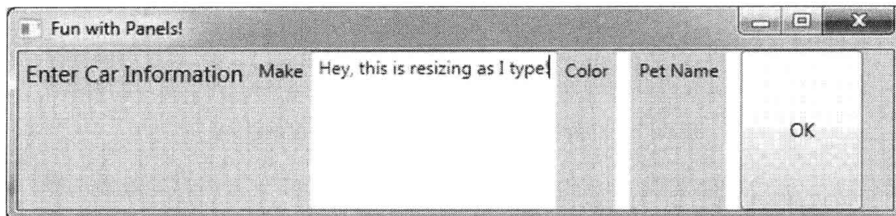


图28-9　水平堆放的内容

与WrapPanel一样，你很少会使用StackPanel直接在窗口中放置内容。StackPanel总是作为主面板的一个子面板出现。

28.3.4　在Grid面板中放置内容

在WPF API提供的所有面板中，Grid是最灵活的一个。和HTML表格类似，Grid可以被划分为一组单元格，每个单元格中都可以放置内容。在定义Grid时，要执行以下3个步骤。

(1) 定义并配置列。

(2) 定义并配置行。

(3) 使用附加属性语法设置每个单元格中的内容。

说明　如果不定义任何行或列，<Grid>默认为一个单元格，包含窗口的整个界面。此外，如果不为<Grid>中的某个子元素设置单元格值，将自动附加为第0列第0行。

实现前两个步骤（定义列和行）时，可以使用<Grid.ColumnDefinitions>和<Grid.RowDefinitions>元素，它们分别包含<ColumnDefinition>和<RowDefinition>元素的集合。网格中的每个单元格都是一个真正的.NET对象，因此你可以根据喜好配置每个单元格的外观和行为。

下面是一个<Grid>定义（位于SimpleGrid.xaml文件中），它放置了如图28-10所示的UI内容。

```
<Grid ShowGridLines ="True" Background ="LightSteelBlue">
  <!-- 定义行和列 -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>

  <!-- 现在向网格中的单元格中添加元素 -->
  <Label x:Name="lblInstruction" Grid.Column ="0" Grid.Row ="0"
        FontSize="15" Content="Enter Car Information"/>
  <Button x:Name="btnOK"  Height ="30" Grid.Column ="0"
```

```
        Grid.Row ="0" Content="OK"/>
   <Label x:Name="lblMake" Grid.Column ="1"
       Grid.Row ="0" Content="Make"/>
   <TextBox x:Name="txtMake" Grid.Column ="1"
        Grid.Row ="0" Width="193" Height="25"/>
   <Label x:Name="lblColor" Grid.Column ="0"
       Grid.Row ="1" Content="Color"/>
   <TextBox x:Name="txtColor" Width="193" Height="25"
        Grid.Column ="0" Grid.Row ="1" />

   <!-- 为了让整个示例更有趣点儿，在Pet Name单元格中添加一些颜色 -->
   <Rectangle Fill ="LightGreen" Grid.Column ="1" Grid.Row ="1" />
   <Label x:Name="lblPetName" Grid.Column ="1" Grid.Row ="1" Content="Pet Name"/>
   <TextBox x:Name="txtPetName" Grid.Column ="1" Grid.Row ="1"
        Width="193" Height="25"/>
</Grid>
```

注意，每个元素（包括浅绿色的Rectangle元素）都使用Grid.Row和Grid.Column附加属性将它们本身连接到网格中的某个单元格。默认情况下，单元格的顺序是从左上角开始的，左上角使用Grid.Column="0" rid.Row="0"指定。由于我们的网格定义了4个单元格，那么右下角的单元格就可以用Grid.Column="1" Grid.Row="1"来识别。

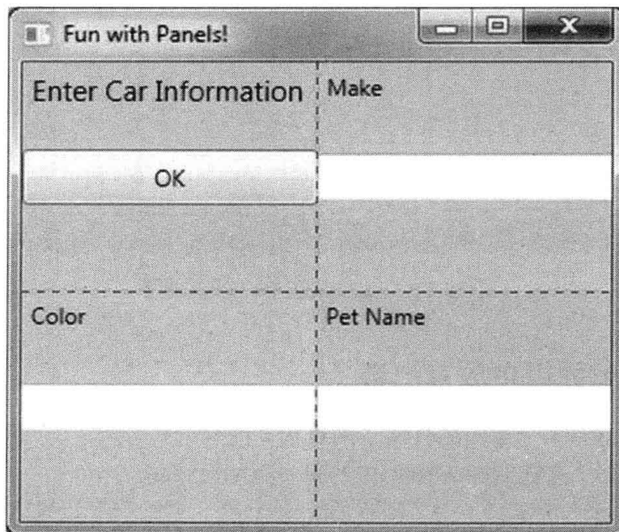


图28-10 Grid面板

使用GridSplitter类型

Grid对象还支持拆分器（splitter），允许用户调整网格类型中行或列的大小。每个可调整大小的单元格都会根据所包含的项进行重塑。向Grid中添加拆分器是很简单的，只需要定义<GridSplitter>控件，并使用附加属性语法指定要拆分的行或列。

注意，你必须指定拆分器的Width或Height值（根据水平拆分或垂直拆分），才能将其显示在屏幕上。考虑下面这个简单的Grid类型，它的拆分器在第一列（Grid.Column="0"）。GridWithSplitter.xaml文件中的内容如下所示：

```
<Grid Background ="LightSteelBlue">
  <!-- 定义列 -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width ="Auto"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>

  <!-- 在第一个单元格中添加这个标签 -->
  <Label x:Name="lblLeft" Background ="GreenYellow"
         Grid.Column="0" Content ="Left!"/>

  <!-- 定义拆分器 -->
  <GridSplitter Grid.Column ="0" Width ="5"/>

  <!--在第二个单元格中添加下面的标签-->
  <Label x:Name="lblRight" Grid.Column ="1" Content ="Right!"/>
</Grid>
```

首先，注意支持拆分器的列的Witdh属性值为Auto。其次，注意<GridSplitter>使用了附加属性语法来定义要应用到哪一列。查看输出结果，可以看到一个5像素的拆分器，它允许你调整每个标签的大小。注意，由于没有指定Label的Height或Width属性，因此它们将充满整个单元格（如图28-11所示）。

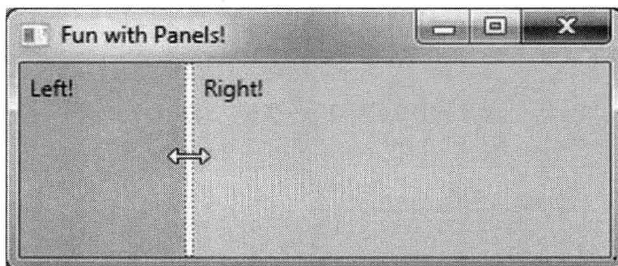


图28-11　包含拆分器的网格类型

28.3.5　在DockPanel面板中放置内容

DockPanel通常作为持有多个面板的容器，来对相关内容进行分组。DockPanel使用附加属性语法（与Canvas和Grid类型类似）来控制DockPanel中的每一项如何停靠。

SimpleDockPanel.xaml文件定义了一个简单的DockPanel，它将产生如图28-12所示的输出结果：

```
<DockPanel LastChildFill ="True">
  <!-- 将各项停靠在面板上 -->
  <Label x:Name="lblInstruction" DockPanel.Dock ="Top"
         FontSize="15" Content="Enter Car Information"/>
  <Label x:Name="lblMake" DockPanel.Dock ="Left" Content="Make"/>
  <Label x:Name="lblColor" DockPanel.Dock ="Right" Content="Color"/>
  <Label x:Name="lblPetName" DockPanel.Dock ="Bottom" Content="Pet Name"/>
  <Button x:Name="btnOK" Content="OK"/>
</DockPanel>
```

说明　如果在DockPanel的同一侧添加了多个元素，它们将在指定的边缘按声明顺序堆放。

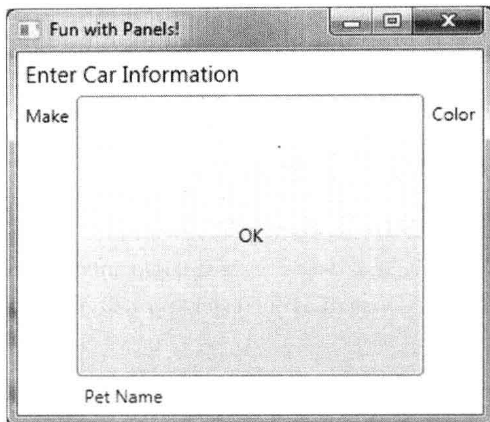


图28-12　简单的DockPanel

使用DockPanel类型的好处是，在用户调整窗口大小时，每个元素仍然能连接到指定的面板边缘(通过DockPanel.Dock)。还要注意<DockPanel>开始标签中将LastChildFill特性设置为true。由于Button控件是容器中最后一个子元素，因此它将在剩余的空间内进行拉伸。

28.3.6　启用Panel类型的滚动功能

值得指出的是，WPF包含ScrollViewer类，为面板对象中的数据提供了自动滚动行为。ScrollViewer.xaml文件的定义如下：

```
<ScrollViewer>
  <StackPanel>
    <Button Content ="First" Background = "Green" Height ="40"/>
    <Button Content ="Second" Background = "Red" Height ="40"/>
    <Button Content ="Third" Background = "Pink" Height ="40"/>
    <Button Content ="Fourth" Background = "Yellow" Height ="40"/>
    <Button Content ="Fifth" Background = "Blue" Height ="40"/>
  </StackPanel>
</ScrollViewer>
```

上面的XAML定义的结果如图28-13所示。

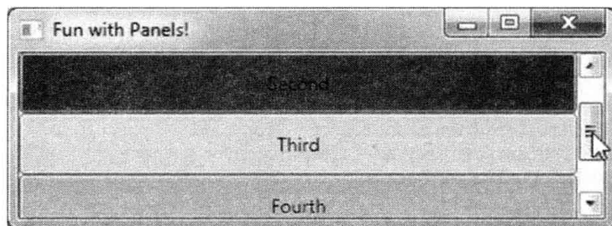


图28-13　使用ScrollViewer类型

正如你期望的，每个面板都提供了多个成员，用于微调内容的位置。很多WPF控件都支持两个属性(Padding和Margin)，面板可以根据这两个属性来放置控件。具体来说，Padding属性控制控件边界

与控件内容之间的空间，Margin控制控件边界与外部容器边界的空间。

我们已经学习了WPF中主要的面板类型，以及它们放置内容的不同方式。接下来，我们将学习如何使用Visual Studio设计器来创建布局。

28.3.7　使用Visual Studio设计器配置Panel

现在我们已经学习了如何用XAML定义常见的布局管理器，你将很高兴看到Visual Studio为构建布局提供了非常好的设计时支持。其关键是本章前面描述的Document Outline窗口。为了演示一些基本概念，创建一个WPF Application项目VisualLayoutTesterApp（该示例仅用来演示布局编辑器，所以我没有放到所提供的源代码中）。

注意，在默认情况下初始的Window使用Grid布局，如下所示：

```
<Window x:Class="VisualLayoutTesterApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

如图28-14所示，如果喜欢使用Grid布局系统，可以用可视布局简单地切割和调整单元格的大小。首先在Document Outline窗口中选择Grid组件，然后点击网格边框来创建新行或新列。

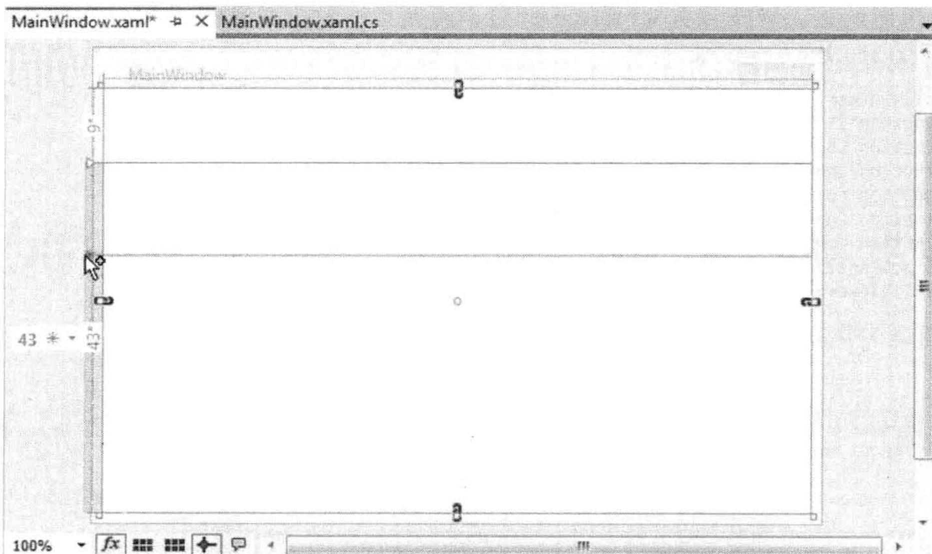


图28-14　使用IDE的设计器可以可视地将Grid控件切割成多个单元格

现在假设我们定义了一个包含多个单元格的网格。我们可以将控件拖放到布局系统的某个单元格中，IDE会自动设置控件的Grid.Row和Grid.Column属性。以下是将一个Button拖放到某预定义的单元格后IDE自动生成的可能的标记：

```
<Button Content="Button" HorizontalAlignment="Left" Margin="10,10,0,0"
        Grid.Row="1" VerticalAlignment="Top" Width="75"
        Grid.Column="1"/>
```

如果不想使用Grid，可以在Document Outline窗口中右键单击布局节点，可以找到允许改变当前容器的菜单选项（如图28-15所示）。要清楚的是，你将（很可能）从根本上改变控件的位置，因为控件要符合新的面板类型的规则。

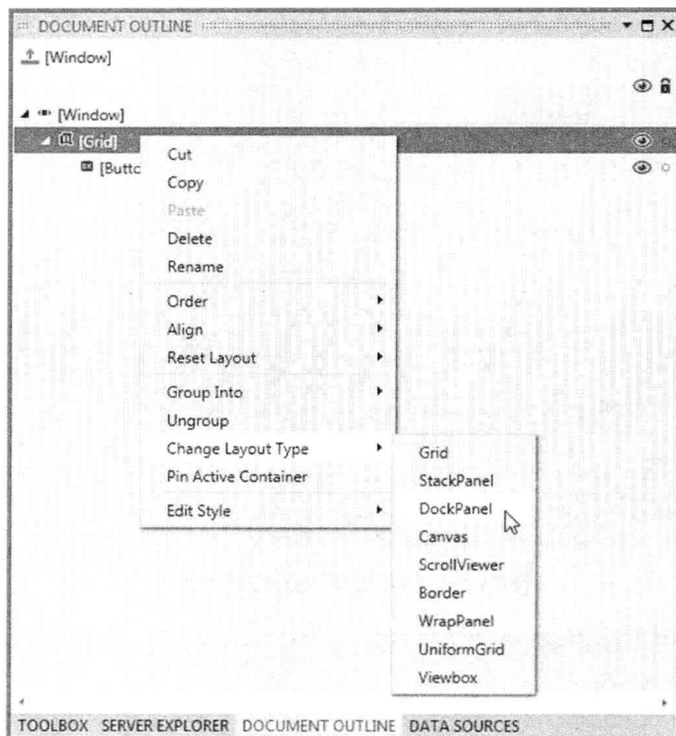


图28-15　Document Outline窗口可以改变面板类型

另一个好用的功能是在可视设计器上选择一组控件，并将它们分组到新的内嵌的布局管理器中。假设我们有一个定义类的一些随机对象的Canvas（如果你想尝试，可以用图28-15所示的技术将初始的Grid转换为Canvas）。现在，在设计器中选择多个项（按住CTRL键并用鼠标左键单击各个项），这时右键单击选中的项，就可以将它们分组到新的子面板中（如图28-16所示）。

然后，再次观察Document Outline窗口来验证嵌套的布局系统。在构建全功能的WPF窗体时，你很可能总是需要使用嵌套的布局系统，而不是简单地选择为所有UI显示使用单个面板（事实上本书其他WPF示例都是如此）。最后，要知道Document Outline窗口上的所有节点都是可拖放的。例如，如果要将当前Canvas中的某个控件移动到父面板中，就可以像如图28-17所示这么做。

在学习其余WPF章节时，我将在必要的时候指出其他的布局捷径。但要知道的是，花时间实验和测试不同的特性是绝对值得的。为了让我们在正确的方向上前行，本章下一个示例将演示如何为一个自定义的文本处理应用程序（带拼写检查）构建嵌套的布局管理器。

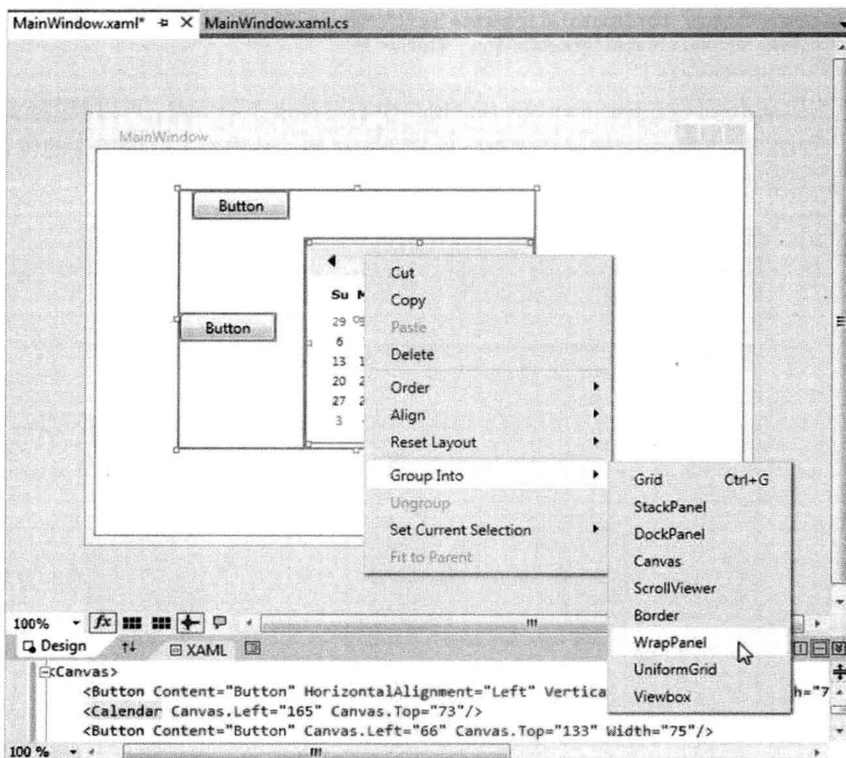


图28-16　将项分组到新的子面板中

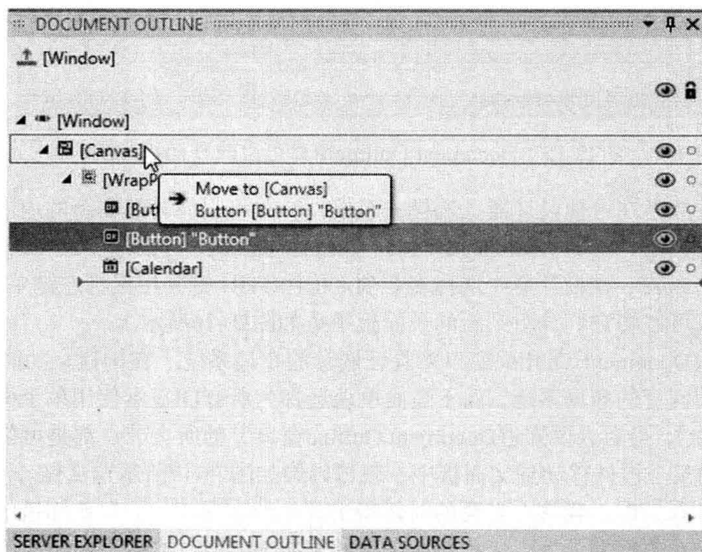


图28-17　通过Document Outline窗口重新放置项

28.4 使用嵌套面板构建窗口框架

如前所述，典型的WPF窗口不会只使用一个面板控件，它们会将面板嵌套使用来得到所需的布局系统。接下来，新建一个WPF Application，取名为MyWordPad。

我们的目标是建立一个布局，使主窗口包含一个顶级菜单系统，菜单系统下面是工具条，窗口下方为状态条。状态条将包含一个面板，当用户选择菜单项（或工具条按钮）时，该面板将显示文字提示。菜单系统和工具条将提供UI触发器来关闭应用程序并在Expander部件中显示拼写建议。图28-18显示了初始的布局，和对XAML的拼写建议。

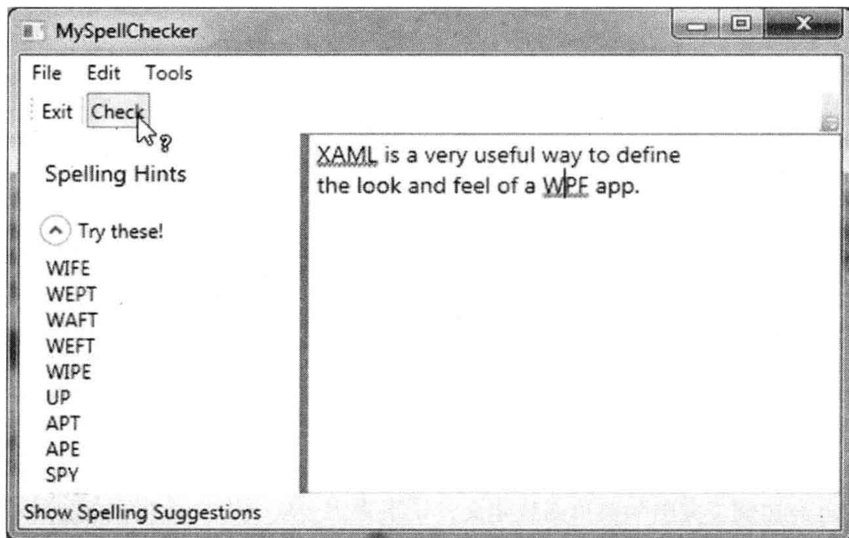


图28-18 使用嵌套面板建立窗口界面

注意，两个工具条按钮不支持图片，而是使用简单的文本值。这对一个产品级别的应用程序来说是不够的，但是为工具条按钮指定图像，通常需要使用嵌套资源，这将在第30章中进行介绍（因此，这里我们使用文本）。同时还要注意，当鼠标放到Check按钮上时，鼠标光标将发生改变，状态条的面板上将显示有用的UI消息。

要构建这样的UI，先更新初始的Window类型的XAML定义，使其使用<DockPanel>子元素，而不是默认的<Grid>，如下所示：

```
<Window x:Class="MyWordPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MySpellChecker" Height="331" Width="508"
  WindowStartupLocation ="CenterScreen">

  <!-- 在该面板中建立窗口的内容 -->
  <DockPanel>
  </DockPanel>

</Window>
```

28.4.1　构建菜单系统

WPF中的菜单系统由包含MenuItem对象集合的Menu类表示。在XAML中构建菜单系统时，可以让每个MenuItem处理不同的事件。最值得注意的事件是Click，它在用户选择子项时发生。在本例中，我们先构建两个顶级菜单项（File和Tools，稍后再构建Edit菜单），它们分别包含Exit和Spelling Hints两个子项。

除了为这两个子项处理Click事件，你还需要处理MouseEnter和MouseExit事件，用来在下一步中设置状态条文本。在<DockPanel>作用域中添加如下标记（可以使用Visual Studio中的Properties窗口来处理各个事件，参见第27章）：

```
<!-- 让菜单系统停靠在上方 -->
<Menu DockPanel.Dock ="Top"
      HorizontalAlignment="Left" Background="White" BorderBrush ="Black">
  <MenuItem Header="_File">
    <Separator/>
    <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
              MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
    </MenuItem>
    <MenuItem Header="_Tools">
      <MenuItem Header ="_Spelling Hints"
          MouseEnter ="MouseEnterToolsHintsArea"
          MouseLeave ="MouseLeaveArea" Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

注意，我们将菜单系统停靠在DockPanel的上方，还使用了<Separator>元素在菜单系统中的Exit选项前面插入一条水平的细线。同时还要注意，每个MenuItem的Header值都嵌入了一个下划线标记（如_Exit）。当用户按下Alt键（快捷键）时，该标记后面的字母将带有下划线。

现在我们已经实现了完整的菜单系统定义，接下来需要实现不同的事件处理程序。首先实现了File→Exit处理程序FileExit_Click()，它将关闭窗口，由于该窗口为顶级窗口，因此也将终止应用程序。每个子项的MouseEnter和MouseExit事件处理程序最终都将更新状态条，但此时我们都先只提供方法外壳。最后是Tools→Spelling Hints菜单项的ToolsSpellingHints_Click()处理程序，我们也先保留外壳。到目前为止，代码隐藏文件中的代码如下：

```
public partial class MainWindow : System.Windows.Window
{
  public MainWindow()
  {
    InitializeComponent();
  }

  protected void FileExit_Click(object sender, RoutedEventArgs args)
  {
    // 关闭该窗口
    this.Close();
  }

  protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
  {
  }
  protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
  {
```

```
    }
    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
    {
    }
    protected void MouseLeaveArea(object sender, RoutedEventArgs args)
    {
    }
}
```

可视地构建菜单

尽管了解如何在XAML中手动定义项总是十分有用的，但这毕竟有些繁琐。Visual Studio为菜单系统、工具条、状态条和很多其他UI控件提供了可视设计支持。举个例子，假设在一个新Window上有一个Menu控件（通过Project→Add Window菜单项插入一个测试窗体）。现在，右键单击Menu控件，会发现一个Add MenuItem选项（如图28-19所示）。

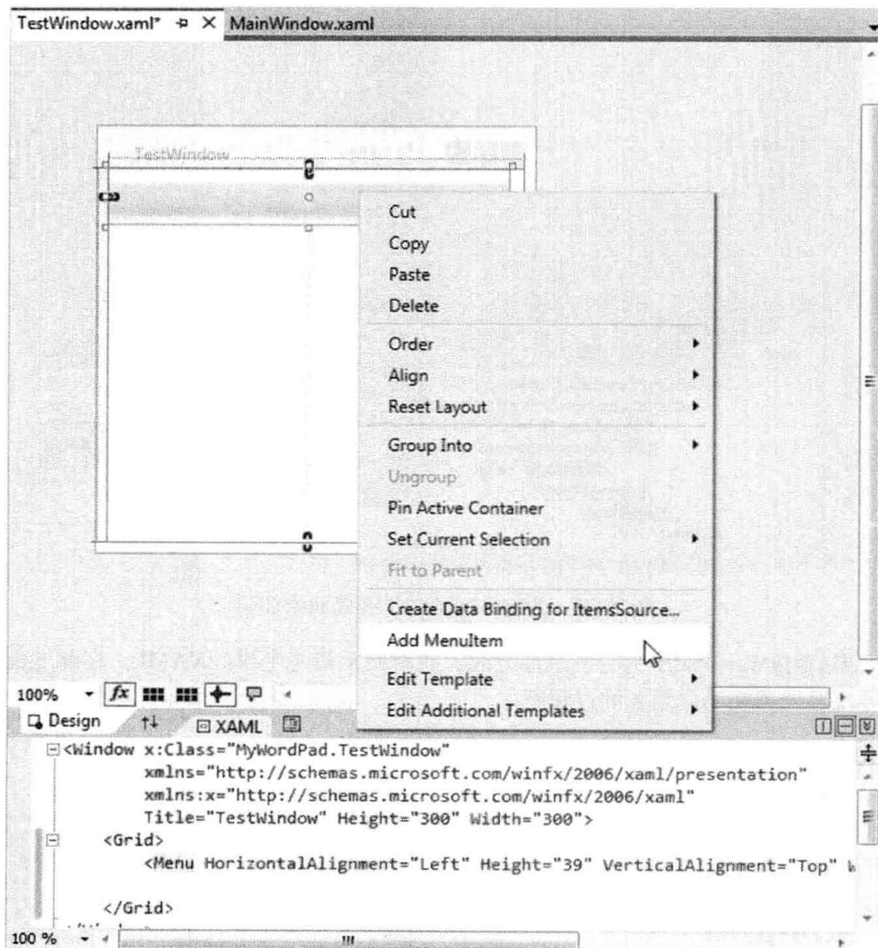


图28-19 为Menu对象可视地添加项

添加完顶级元素之后，可以再添加子菜单项、分割器、展开或折叠菜单，或再次右键单击执行其他与菜单有关的操作。图28-20展示了一个简单菜单系统的可视化设计（请查看生成的XAML）。

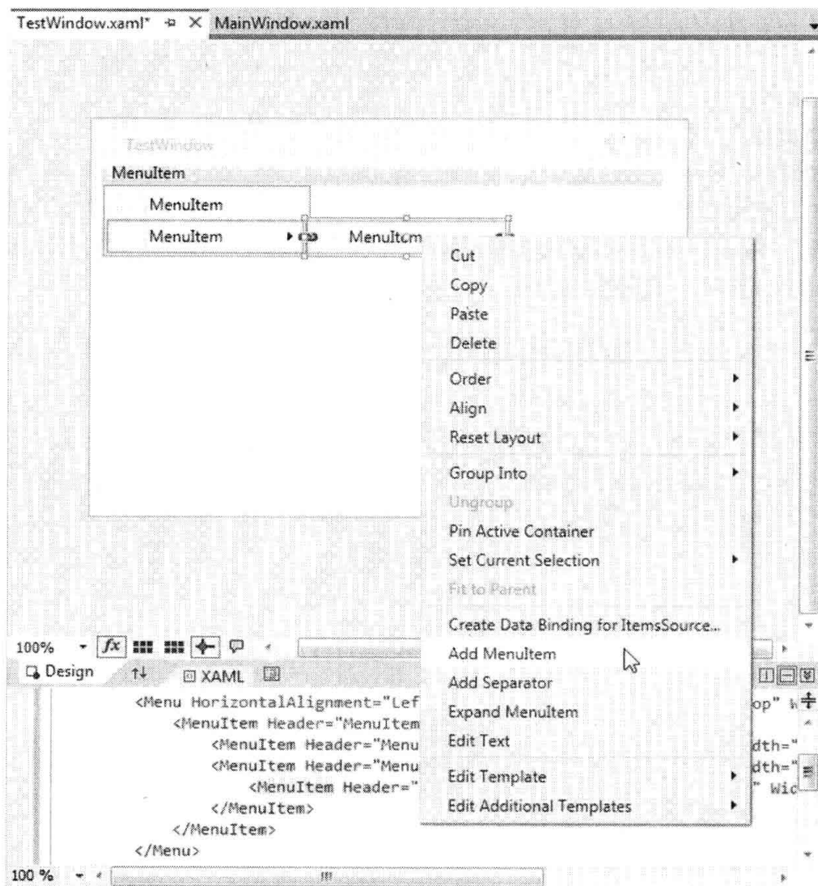


图28-20　为MenuItem对象可视地添加项

在介绍完当前MyWordPad示例的其余部分时，我将展示最终生成的XAML。在此之前，花点时间体验一下用可视设计器来简化任务的过程吧。

28.4.2　构建工具条

工具条（由WPF中的ToolBar类表示）通常提供了另一种激活菜单项的方式。在定义完<Menu>后，直接添加如下的标记：

```
<!-- 在菜单下面放置工具条 -->
<ToolBar DockPanel.Dock ="Top" >
  <Button Content ="Exit" MouseEnter ="MouseEnterExitArea"
          MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
  <Separator/>
```

```
  <Button Content ="Check" MouseEnter ="MouseEnterToolsHintsArea"
          MouseLeave ="MouseLeaveArea" Click ="ToolsSpellingHints_Click"
          Cursor="Help" />
</ToolBar>
```

我们的ToolBar控件包含两个Button控件，它们与前面的菜单处理相同的事件，并且由代码文件中相同的方法进行处理。这样处理程序就可以同时为菜单项和工具条按钮服务。尽管这个工具条只使用了典型的控制按钮，但你应该明白ToolBar类型“是一个”ContentControl，因此可以在其表面上嵌入任何类型（如下拉框、图像和图形）。除此之外，唯一有趣的地方是Check按钮通过Cursor属性自定义了鼠标光标。

说明 你可以将ToolBar元素包裹到<ToolBarTray>元素内，<ToolBarTray>元素控制一组ToolBar对象的布局、停靠和拖曳操作。详细信息请参考.NET Framework 4.5 SDK文档。

28.4.3 构建状态条

<DockPanel>下方将放置一个StatusBar控件，它包含一个本章前面没有介绍过的<TextBlock>控件。TextBlock可用来保存支持各种文字批注的文本，如加粗文本、下划线文本、换行符等。直接在前面ToolBar定义的后面添加如下标记：

```
<!-- 在底部放置一个状态条 -->
<StatusBar DockPanel.Dock ="Bottom" Background="Beige" >
  <StatusBarItem>
    <TextBlock Name="statBarText" Text="Ready"/>
  </StatusBarItem>
</StatusBar>
```

28.4.4 完成UI设计

UI设计的最后一步是定义一个可拆分为两列的Grid。在其左半边放置一个<StackPanel>，它包含一个Expander控件，显示拼写提示列表。在其右半边放置一个TextBox控件，支持多行和滚动条，并且启用拼写检查。我们将整个<Grid>停靠在父元素<DockPanel>的左边。在描述StatusBar的标记后面直接添加如下的XAML标记，并完成整个窗口UI的设计：

```
<Grid DockPanel.Dock ="Left" Background ="AliceBlue">
  <!-- 定义行和列 -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <GridSplitter Grid.Column ="0" Width ="5" Background ="Gray" />
  <StackPanel Grid.Column="0" VerticalAlignment ="Stretch" >
    <Label Name="lblSpellingInstructions" FontSize="14" Margin="10,10,0,0">
     Spelling Hints
    </Label>

    <Expander Name="expanderSpelling" Header ="Try these!"
              Margin="10,10,10,10">
```

```
      <!-- 将以编程方式填充 -->
      <Label Name ="lblSpellingHints" FontSize ="12"/>
    </Expander>
  </StackPanel>

  <!-- 输入文字的区域 -->
  <TextBox  Grid.Column ="1"
            SpellCheck.IsEnabled ="True"
            AcceptsReturn ="True"
            Name ="txtData"  FontSize ="14"
            BorderBrush ="Blue"
            VerticalScrollBarVisibility="Auto"
            HorizontalScrollBarVisibility="Auto">
  </TextBox>
</Grid>
```

28.4.5　实现MouseEnter/MouseLeave事件处理程序

此时我们完成了窗口的UI，剩下的任务就是实现遗留的事件处理程序。我们先更新C#代码文件，使所有MouseEnter和MouseLeave处理程序都在状态条的文本窗格中显示恰当的消息来帮助用户，如下所示：

```
public partial class MainWindow : System.Windows.Window
{
...
  protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
  {
    statBarText.Text = "Exit the Application";
  }
  protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
  {
    statBarText.Text = "Show Spelling Suggestions";
  }
  protected void MouseLeaveArea(object sender, RoutedEventArgs args)
  {
    statBarText.Text = "Ready";
  }
}
```

此时运行应用程序，可以看到状态条的文本会根据鼠标悬停的菜单项/工具条按钮而改变。

28.4.6　实现拼写检查逻辑

WPF API发布时内置了对于拼写检查的支持，它独立于微软Office产品。这意味着你不需要通过COM互操作层来使用Word中的拼写检查器，而只要简单地添加几行代码就可以得到同样类型的支持。

你也许还记得我们在定义<TextBox>控件时，将SpellCheck.IsEnabled属性设置为true。这样，拼错的单词将用红色波浪线标注，就像在Office里那样。更妙的是，编程模型还可以访问拼写检查器引擎，为拼错的单词提供更改建议。将下面的代码添加到ToolsSpellingHints_Click()方法中：

```
protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
{
  string spellingHints = string.Empty;

  // 获取当前位置的拼写错误
```

```
    SpellingError error = txtData.GetSpellingError(txtData.CaretIndex);
    if (error != null)
    {
      // 构建更改建议的字符串
      foreach (string s in error.Suggestions)
      {
        spellingHints += string.Format("{0}\n", s);
      }

      // 显示建议和扩展Expander
      lblSpellingHints.Content = spellingHints;
      expanderSpelling.IsExpanded = true;
    }
  }
```

上面的代码非常简单。我们使用CaretIndex属性得到文本框中当前光标所在的位置，提取SpellingError对象。如果这个位置存在错误（即值不为null），则循环Suggestions属性中的建议列表。得到错误单词的所有更改建议后，将其连接到Expander中的Label。

这样就可以了。只需要几行程序代码（和少量XAML），就完成了文字处理器的开端。如果理解了控件命令，就可以再添加一些时髦的特色。

28.5 WPF 命令

WPF通过命令架构提供了一种控件无关的事件。常见的.NET事件定义在指定的基类中，并且只能由该类或该类的派生类使用。因此，普通的.NET事件与其所在的类是紧紧耦合在一起的。

相反，WPF命令是与事件类似的实体，但它们与具体的控件无关，并且在大多数情况下可应用于多种（并且看上去没有关联的）控件类型。例如，WPF支持Copy、Paste和Cut命令，可应用于各种各样的UI元素（如菜单项、工具条按钮和自定义按钮）和快捷键（如Ctrl+C组合键和Ctrl+V组合键）。

尽管其他UI工具（如Windows Forms）也为上述功能提供了标准事件，但却常常会带来冗余的、难以维护的代码。在WPF模型下，我们可以使用命令，通常最终将得到简洁灵活的代码库。

28

28.5.1 内置的命令对象

WPF发布了一些内置的控件命令，它们都可以配置相关的快捷键（或其他输入方式）。从编程角度来说，WPF命令是任意支持属性（通常以Command为后缀）的对象，能够返回实现了ICommand接口的对象，如下所示：

```
public interface ICommand
{
  // 当影响命令能否执行的变化发生时，触发该事件
  event EventHandler CanExecuteChanged;

  // 返回判断当前状态下命令能否执行的方法
  bool CanExecute(object parameter);

  // 定义命令触发时将调用的方法
  void Execute(object parameter);
}
```

WPF提供了各种命令类（大约100个可用的命令对象），这些类定义了大量公开特定命令对象的属

性，而每一个对象都实现了ICommand。表28-3列出了一些可用的标准命令对象（详细内容可参考.NET Framework 4.5 SDK文档）。

表28-3　内置的WPF控件命令对象

WPF类	命令对象	作　用
ApplicationCommands	Close、Copy、Cut、Delete、Find、Open、Paste、Save、SaveAs、Redo、Undo	应用程序级别的各种命令
ComponentCommands	MoveDown、MoveFocusBack、MoveLeft、MoveRight、ScrollToEnd、ScrollToHome	UI组件常用的各种命令
MediaCommands	BoostBase、ChannelUp、ChannelDown、FastForward、NextTrack、Play、Rewind、Select、Stop	多媒体相关的各种命令
NavigationCommands	BrowseBack、BrowseForward、Favorites、LastPage、NextPage、Zoom	WPF导航模型相关的各种命令
EditingCommands	AlighCenter、CorrectSpellingError、DecreaseFontSize、EnterLineBreak、EnterParagraphBreak、MoveDownByLine、MoveRightByWord	WPF Documents API相关的各种命令

28.5.2　将命令连接到Command属性

将WPF命令属性连接到支持Command属性的UI元素（如Button或MenuItem）是非常简单的。更新当前的菜单系统，使其支持一个新的顶级菜单Edit和3个分别表示复制、粘贴和剪切文本数据的子项，如下所示：

```
<Menu DockPanel.Dock ="Top"
      HorizontalAlignment="Left"
      Background="White" BorderBrush ="Black">
  <MenuItem Header="_File" Click ="FileExit_Click" >
    <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
              MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
  </MenuItem>

  <!-- 使用命令的新菜单项 -->
  <MenuItem Header="_Edit">
    <MenuItem Command ="ApplicationCommands.Copy"/>
    <MenuItem Command ="ApplicationCommands.Cut"/>
    <MenuItem Command ="ApplicationCommands.Paste"/>
  </MenuItem>

  <MenuItem Header="_Tools">
    <MenuItem Header ="_Spelling Hints"
              MouseEnter ="MouseEnterToolsHintsArea"
              MouseLeave ="MouseLeaveArea"
              Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

注意，Edit菜单的每个子项都为Command属性设置了值。这意味着菜单项将自动接收正确的名称和快

捷键（如Ctrl+X为剪切操作），即现在应用程序可以不添加任何程序代码就支持复制、剪切和粘贴操作。

运行应用程序并选择一段文本，即可使用新的菜单项了。此外，应用程序还可以响应标准的右键操作，为用户呈现同样的选项（如图28-21所示）。

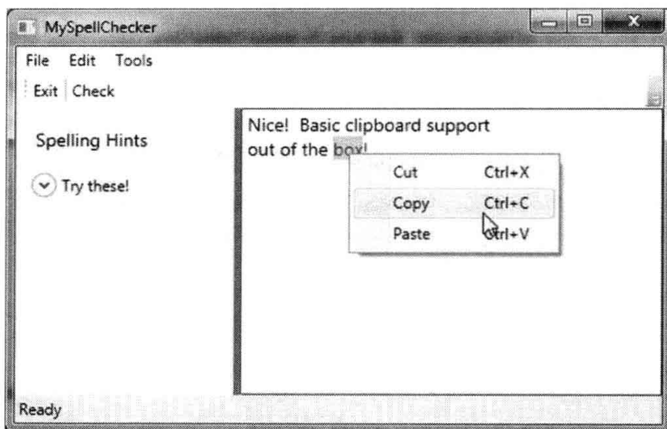


图28-21　命令对象内嵌了大量的功能

28.5.3　将命令连接到任意行为

如果希望将命令对象连接到任意（应用程序特定的）事件，就需要编写一些程序代码了。这并不复杂，但确实包含了比XAML更多的逻辑。例如，假设希望整个窗口能够响应F1键，当用户按下F1键时，将激活相关的帮助系统。同样，假设主窗口的代码文件定义了新的方法`SetF1CommandBinding()`，在构造函数调用完`InitializeComponent()`方法之后，会调用这个方法：

```
public MainWindow()
{
  InitializeComponent();
  SetF1CommandBinding();
}
```

这个新方法将以编程方式新建`CommandBinding`对象，无论何时你都可以使用该对象将命令对象绑定到应用程序中给定的事件处理程序。这里，我们配置`CommandBinding`对象，使其操作`ApplicationCommands.Help`命令，该命令自动响应F1键：

```
private void SetF1CommandBinding()
{
  CommandBinding helpBinding = new CommandBinding(ApplicationCommands.Help);
  helpBinding.CanExecute += CanHelpExecute;
  helpBinding.Executed += HelpExecuted;
  CommandBindings.Add(helpBinding);
}
```

大多数`CommandBinding`对象都需要处理`CanExecute`事件（指定命令是否会响应程序操作）和`Executed`事件（编写命令响应之后会发生的内容）。在`Window`派生类型中添加如下的事件处理程序（注意每个方法的格式与相关的委托所要求的格式相同）：

```
private void CanHelpExecute(object sender, CanExecuteRoutedEventArgs e)
{
  // 如果要阻止命令执行，可以将CanExecute设置为false
  e.CanExecute = true;
}

private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
  MessageBox.Show("Look, it is not that difficult. Just type something!",
                  "Help!");
}
```

在上面的代码片段中，我们实现了CanHelpExecute()，简单地返回true，始终允许启动F1帮助。但如果某些情况下不希望显示帮助系统，可以返回false。HelpExecuted()中所显示的“帮助系统”仅仅是一个消息框。这时运行应用程序，当按下键盘上的F1键时，可以看到（说句不好听的，是什么忙也帮不上的）用户帮助系统（如图28-22所示）。

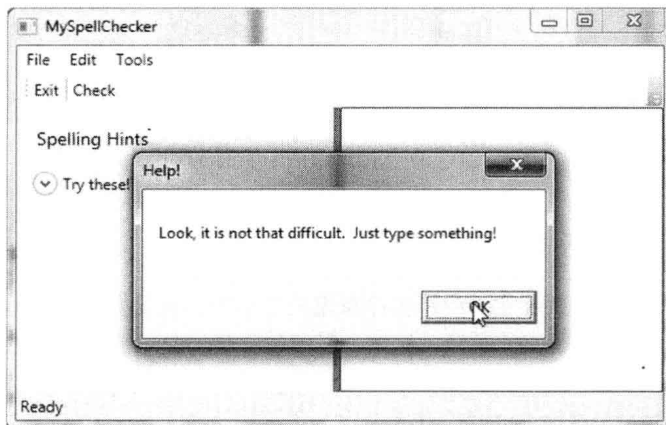


图28-22　自定义的帮助系统（可能没什么帮助）

28.5.4　使用Open和Save命令

在当前示例的最后，我们将添加将文本数据保存为外部文件以及打开*.txt文件进行编辑的功能。如果想绕远路，你可以手工添加编程逻辑来根据TextBox中是否包含数据来启用或禁用新菜单项。但实际上，我们还是可以使用命令来减轻负担。

更新表示顶级File菜单的<MenuItem>元素，添加两个新的使用Save和Open ApplicationCommands对象的子菜单：

```
<MenuItem Header="_File">
  <MenuItem Command ="ApplicationCommands.Open"/>
  <MenuItem Command ="ApplicationCommands.Save"/>
  <Separator/>
  <MenuItem Header ="_Exit"
            MouseEnter ="MouseEnterExitArea"
            MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>

</MenuItem>
```

回忆一下，所有实现ICommand接口的对象都定义了两个事件（CanExecute和Executed）。现在我们需要启用整个窗口，来验证当前是否能够触发这些命令。如果能，就可以定义事件处理程序来执行自定义代码。

要实现上述操作，需要填充窗口中维护的CommandBindings集合。在XAML中，可以使用属性元素语法来定义<Window.CommandBindings>作用域，并在其中放置<CommandBinding>定义。将<Window>更新如下：

```
<Window x:Class="MyWordPad.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MySpellChecker" Height="331" Width="508"
  WindowStartupLocation ="CenterScreen" >

  <!--在测试Open和Save命令时，通知Window调用哪个处理程序 -->
  <Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
                    Executed="OpenCmdExecuted"
                    CanExecute="OpenCmdCanExecute"/>
    <CommandBinding Command="ApplicationCommands.Save"
                    Executed="SaveCmdExecuted"
                    CanExecute="SaveCmdCanExecute"/>
  </Window.CommandBindings>

  <!-- 该面板建立窗口的内容 -->
  <DockPanel>
  ...
  </DockPanel>
</Window>
```

现在，在XAML编辑器中右击Executed和CanExecute特性，选择Navigate to Event Handler菜单选项。我们在第27章中介绍过，这样可以为事件自动生成代码存根。现在，在当前窗口的C#代码文件中包含4个空的处理程序。

CanExecute事件处理程序设置了传入的CanExecuteRoutedEventArgs对象的CanExecute属性，通知窗口可以随时触发相应的Executed事件：

```
private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = true;
}

private void SaveCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = true;
}
```

相应的Executed处理程序执行显式打开和保存对话框的实际工作，以及将TextBox中的数据发送到文件中。要确保在代码文件中引入了System.IO和Microsoft.Win32命名空间。完整的代码是十分简单的：

```
private void OpenCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
  // 创建打开文件的对话框，并且只显示XAML文件
  OpenFileDialog openDlg = new OpenFileDialog();
  openDlg.Filter = "Text Files |*.txt";

  // 是否单击了OK按钮
```

```
        if (true == openDlg.ShowDialog())
        {
          // 加载所选文件的所有文本
          string dataFromFile = File.ReadAllText(openDlg.FileName);

          // 在TextBox中显示字符串
          txtData.Text = dataFromFile;
        }
      }

      private void SaveCmdExecuted(object sender, ExecutedRoutedEventArgs e)
      {
        SaveFileDialog saveDlg = new SaveFileDialog();
        saveDlg.Filter = "Text Files |*.txt";

        // 是否单击了OK按钮
        if (true == saveDlg.ShowDialog())
        {
          // 将TextBox中的数据保存到命名的文件中
          File.WriteAllText(saveDlg.FileName, txtData.Text);
        }
      }
```

至此，本示例以及对于WPF控件的初步学习就结束了。我们学习了如何使用基础命令、菜单系统、状态条、工具条、嵌套面板和一些基础的UI控件，如TextBox和Expander。在接下来的示例中，我们将使用一些更加特殊的控件，同时研究几个重要的WPF服务。

源代码　MyWordPad项目的源代码位于Chapter 28子目录下。

28.6　深入了解 WPF API 和控件

本章剩余部分讲解如何使用Visual Studio创建崭新的WPF。我们的目的是创建由包含一组选项卡的TabControl微件组成的用户界面。每一个标签都会阐明几个新WPF控件和你可能想在自己的软件项目中使用的API。在这个过程中，你还能学到Visual Studio WPF设计器的其他功能。

使用TabControl

首先，创建一个新的WPF应用程序WpfControlsAndAPI。如前所示，初始窗口将包含一个含有4个不同选项卡的TabControl，每个选项卡都包含一些相关的控件和WPF API。在Visual Studio Toolbox中找到TabControl控件。拖放一个控件到设计器上，使其占据界面的大部分区域。将这个UI元素重命名为myTabSystem。

这时你应该注意到系统自动给出了两个选项卡项。要添加其他选项卡，只需要右键单击Document Outline窗口中的TabControl节点，选择Add TabItem菜单选项（也可以右键单击设计器上的TabControl激活该菜单选项）。两种方法均可添加更多选项卡（图28-23显示了Document Outline方法）。

然后，选中所有TabItem控件（在设计器上或者通过Document Outline窗口），更改每个选项卡的Header属性，分别输入为Ink API、Documents、Data Binding和DataGrid。此时，窗口设计器应该如图28-24所示。

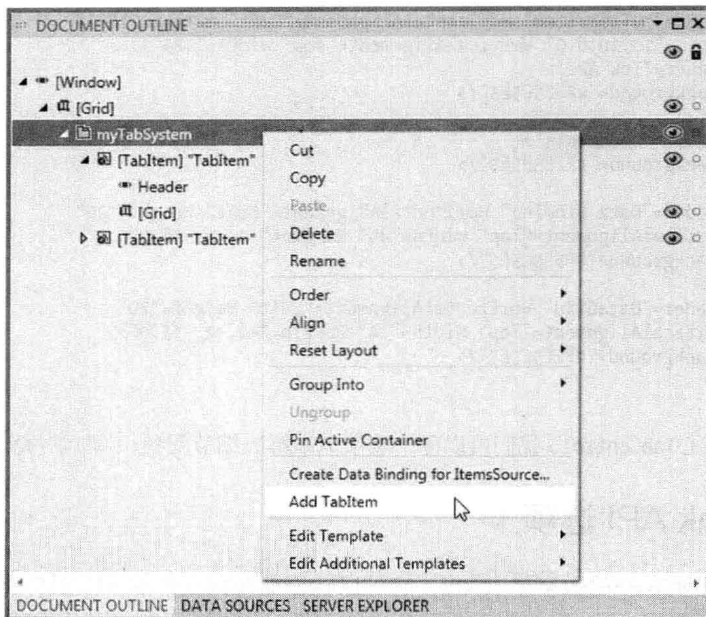


图28-23　可视地添加TabItem

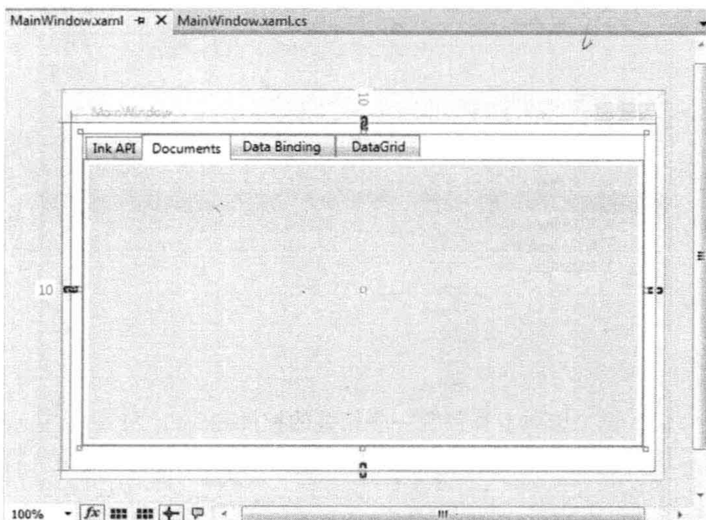


图28-24　选项卡系统的原始布局

现在，再次单击各个选项卡，使用Properties窗口为它们取一个唯一的、恰当的名称。注意，不管使用哪种方法选取，都将激活该选项卡，并且可以通过拖曳Toolbox窗口中的控件来设计该选项卡。在设计之前，先单击XAML按钮查看IDE生成的XAML，标记将与下面的类似（标记将根据设置的属性的不同而有所区别）：

```
<TabControl x:Name="myTabSystem" HorizontalAlignment="Left" Height="280"
            Margin="10,10,0,0" VerticalAlignment="Top" Width="489">
    <TabItem Header="Ink API">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
    <TabItem Header="Documents">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
    <TabItem Header="Data Binding" HorizontalAlignment="Left" Height="20"
             VerticalAlignment="Top" Width="95" Margin="-2,-2,-36,0">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
    <TabItem Header="DataGrid" HorizontalAlignment="Left" Height="20"
             VerticalAlignment="Top" Width="74" Margin="-2,-2,-15,0">
        <Grid Background="#FFE5E5E5"/>
    </TabItem>
</TabControl>
```

现在我们有了核心TabControl，我们可以逐个选项卡来解决细节问题，同时学习WPF API的更多特性。

28.7　构建 Ink API 选项卡

第一个选项卡显示WPF的数字墨水API，它可以轻松地在程序中加入绘图功能。当然，应用程序没有必要一定是绘图程序，该API可用于各种各样的需求，如捕获平板电脑手写笔的输入。

首先在Document Outline区域中找到代表Ink API选项卡的节点并展开它。可以看到该TabItem默认的布局管理器为<Grid>。右击该布局管理器，将其改为StackPanel（如图28-25所示）。

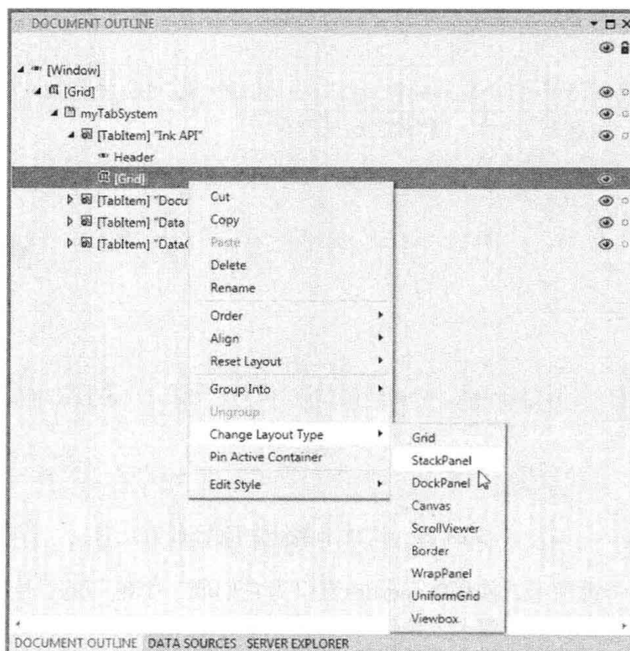


图28-25　更改第一卡选项卡项的布局管理器

28.7.1 设计工具条

在Document Outline编辑器中选中StackPanel节点，并插入一个新的ToolBar控件inkToolbar。然后选中inkToolbar节点，将Toolbar控件的Height设置为60（使用默认Width值即可）。找到Properties窗口的Common部分，单击Items(Collection)属性的椭圆形按钮（如图28-26所示）。

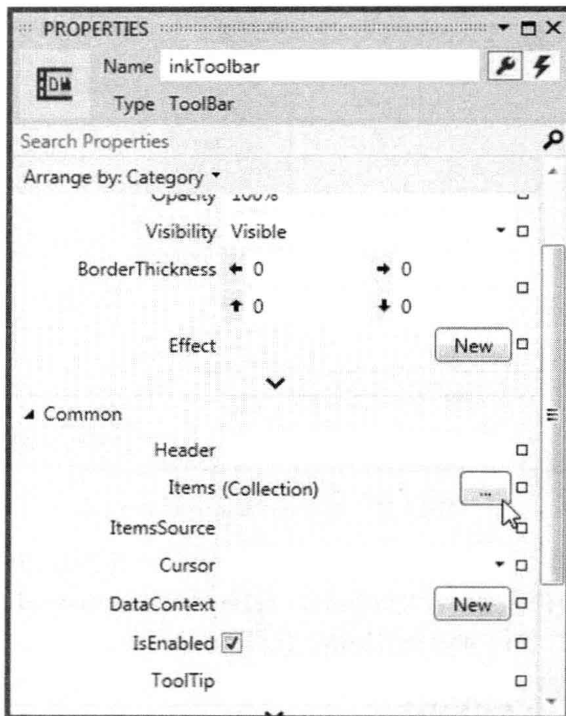


图28-26 使用items编辑器填充ToolBar

单击该按钮后，弹出一个对话框，允许我们选择希望添加到ToolBar中的控件。单击对话框底部中央的下拉列表框，添加3个RadioButton控件。可以使用该对话框内嵌的Properties编辑器，将每个RadioButton的Height设置为50，Width设置为100（这些属性同样位于Layout区域）。同样，将每个RadioButton的Content属性（位于Common区域）设置为Ink Mode!、Erase Mode!和Select Mode!（如图28-27所示）。

添加完3个RadioButton控件后，再使用Items编辑器下拉列表添加一个Separator控件。最后，需要添加一个ComboBox（不是ComboBoxItem）控件。在使用Items对话框插入非标准控件时，从下拉列表中选择<Other Type...>选项。这将打开Select Object编辑器，可以输入所需控件的名称。确保勾选了所有程序集选项，然后就可以搜索你感兴趣的控件了（如图28-28所示）。

将ComboBox的Width属性设置为100，并在属性编辑器的Common部分（再次）使用Items (Collection)属性向ComboBox控件中添加3个ComboBoxItem对象。将各个ComboBoxItem的Content属性分别设置为Red、Green和Blue。

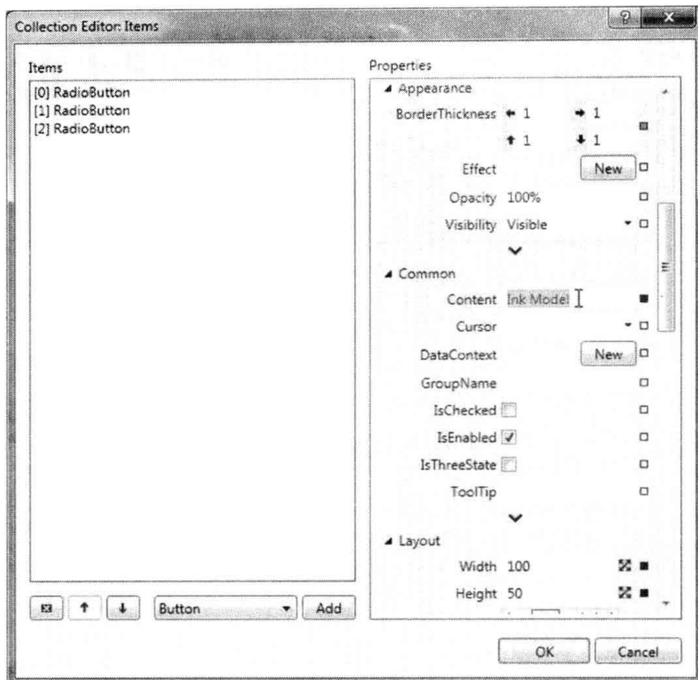


图28-27 设置各个RadioButton

完成之后，关闭编辑器，回到窗口设计器。本节的最后一个任务是使用Name属性为各个项指定变量名。将3个RadioButton控件分别命名为inkRadio、selectRadio和eraseRadio。同样，将ComboBox控件命名为comboColors。第一个控件TabItem的XAML看起来如下所示：

```
<TabItem Header="Ink API">
  <StackPanel Background="#FFE5E5E5">
    <ToolBar x:Name="inkToolbar" HorizontalAlignment="Left" Width="479" Height="60">
      <RadioButton x:Name="inkRadio" Content="Ink Mode!" Height="50" Width="100"/>
      <RadioButton x:Name="selectRadio" Content="Erase Mode!" Height="50" Width="100"/>
      <RadioButton x:Name="eraseRadio" Content="Select Mode!" Height="50" Width="100"/>
      <Separator/>
      <ComboBox x:Name="comboColors" Width="100">
        <ComboBoxItem Content="Red"/>
        <ComboBoxItem Content="Green"/>
        <ComboBoxItem Content="Blue"/>
      </ComboBox>
    </ToolBar>
  </StackPanel>
</TabItem>
```

说明 由于我们是使用IDE构建工具条的，你可能会认为如果简单地手工编辑XAML，将会更快地完成任务。如果你对直接输入标记非常熟练，你当然可以直接输入。不过强烈建议你花点儿时间多多练习Visual Studio WPF Properties编辑器。用久了你会发现，这个编辑器中有大量高级特性。

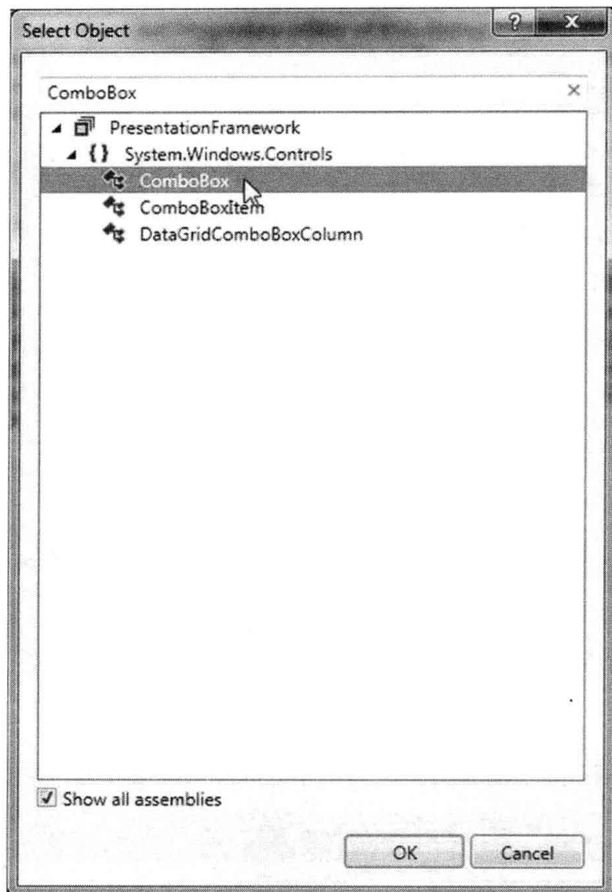


图28-28 使用Select Object编辑器

28.7.2 RadioButton控件

本例中的3个RadioButton是互斥的。在其他GUI框架中，要想让一组相关控件（如单选按钮）互斥，需要将它们放在同一个组合框中。在WPF中不必这么做。你可以简单地将它们设置为同样的分组名称。这样相关的项就没有必要放置在相同的物理区域，而是可以放在窗口的任何位置。

在设计器中选中各个RadioButton（按住Shift键单击），将GroupName属性（位于Properties窗口的Common Properties区域）设置为InkMode。

如果RadioButton控件没有放置在父面板控件中，将和Button控件呈现为相同的UI。但与Button不同的是，RadioButton类包含IsChecked属性，用户单击这个UI元素时，该属性将在true和false之间切换。此外，RadioButton还提供了两个事件（Checked和Unchecked），可以拦截这种状态变化。

如果想使RadioButton控件的外观与常见的单选按钮有所不同，可以按住Shift键单击各个控件，然后用右键单击，选择Group Into→Border菜单选项（如图28-29所示）。

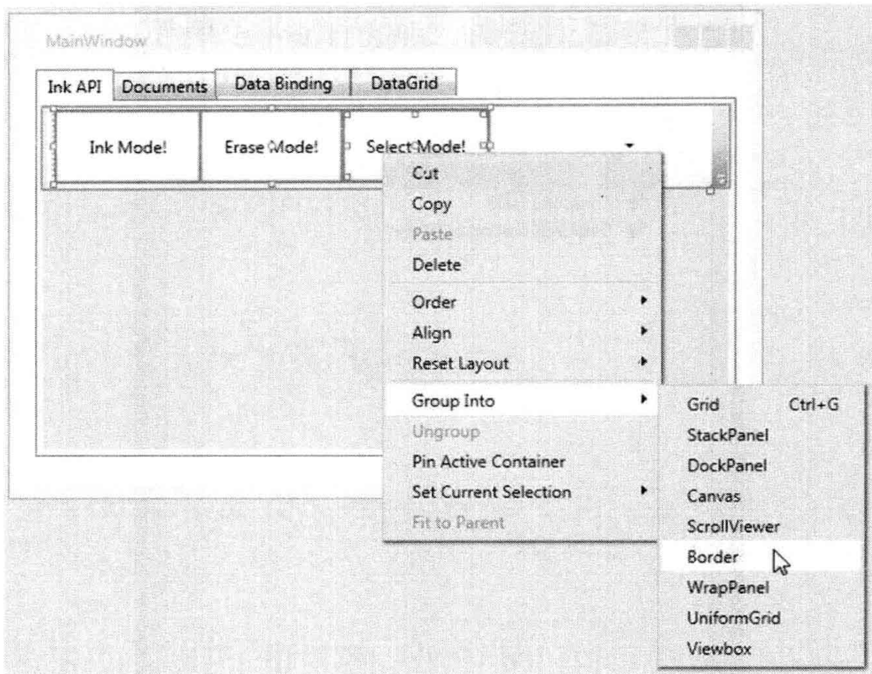


图28-29　在Border控件中进行分组

这时，按下F5键测试程序。你将看到3个互斥的单选按钮和包含3个选项的组合框（如图28-30所示）。

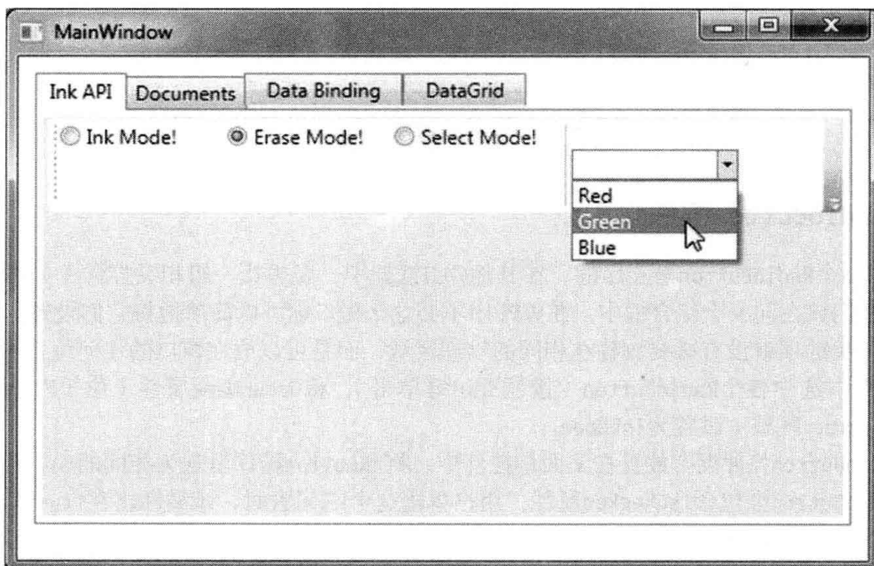


图28-30　完整的工具条系统

28.7.3 处理Ink API选项卡的事件

Ink API选项卡的下一步是为每个RadioButton控件处理Click事件。与本书中的其他WPF项目类似，选中Visual Studio Properties编辑器中包含一个闪电图标的按钮即可输入事件处理程序的名称。用该方法将所有单选按钮的Click事件都指向同一个处理程序RadioButtonClicked。处理完3个Click事件，接下来处理ComboBox的SelectionChanged事件，将处理程序的名称命名为ColorChanged。完成之后的C#代码如下：

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    this.InitializeComponent();

    // 在此插入创建对象所需的代码
  }
  private void RadioButtonClicked(object sender, System.Windows.RoutedEventArgs e)
  {
    // TODO: 在此添加事件处理程序的实现
  }

  private void ColorChanged(object sender,
    System.Windows.Controls.SelectionChangedEventArgs e)
  {
    // TODO: 在此添加事件处理程序的实现
  }
}
```

稍后我们将实现这些处理程序，现在先保留为空实现。

28.7.4 InkCanvas控件

为了完成该选项卡的UI，我们需要在StackPanel中放置一个InkCanvas，使其显示在刚刚创建的Toolbar下面。默认情况下，Visual Studio Toolbox不会显示所有可能的WPF组件。不过你可以简单地输入必要的XAML，需要知道的是，你自己可以更新显示在Toolbox中的项。

我们来看一下如何操作。右键单击Toolbox区域的任意位置，选择Choose Items...菜单选项。稍等片刻，你就能看到待添加到Toolbox的所有组件。从使用情况看，我们感兴趣的是添加InkCanvas控件（如图28-31所示）。

在Document Outline编辑器中选中tabInk对象的StackPanel，添加一个InkCanvas，取名为myInkCanvas。调整新控件大小，使其占据选项卡的大部分区域。你还可以使用Brushes编辑器为InkCanvas设置背景色（下一章将详细介绍画刷编辑器）。然后按F5键。你将看到在单击并拖曳鼠标左键的时候，已经可以在画布上绘制数据了（如图28-32所示）。

InkCanvas不仅能绘制鼠标（或手写笔）笔画，还支持大量由EditingMode属性控制的编辑模式。该属性可以设置为InkCanvasEditingMode枚举的任何值。本例使用Ink模式，这也是默认的选项。Select模式允许用户用鼠标选择一个区域进行移动或调整大小。而EraseByStroke将删除之前的鼠标笔画。

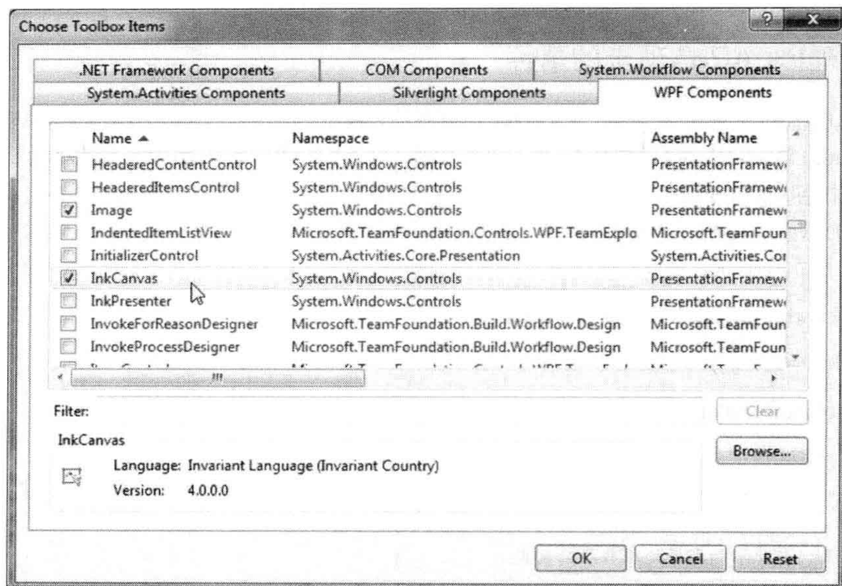


图28-31　为Visual Studio Toolbox添加新组件

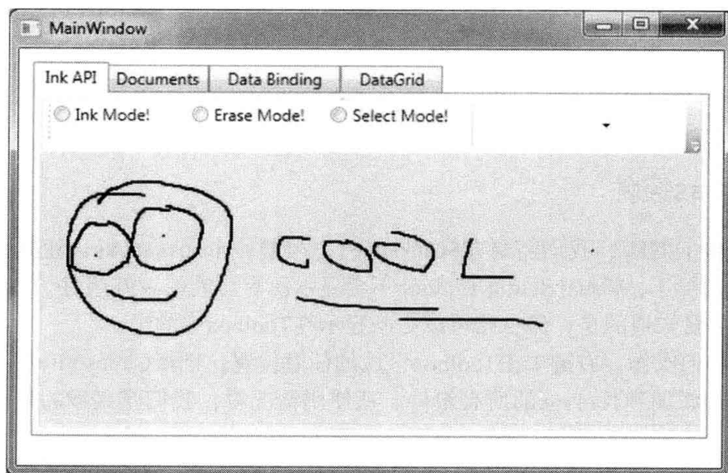


图28-32　InkCanvas实战演练

说明　笔画是一次鼠标按下/抬起操作时呈现的内容。InkCanvas将所有笔画存储在一个Stroke-Collection对象中，该对象可使用Strokes属性访问。

用下面的逻辑更新RadioButtonClicked()处理程序，该程序将根据所选的RadioButton为InkCanvas设置正确的模式：

```
private void RadioButtonClicked(object sender,
  System.Windows.RoutedEventArgs e)
{
  // 根据传入的按钮，设置InkCanvas的操作模式
  switch((sender as RadioButton).Content.ToString())
  {
    // 这些字符串必须与各个RadioButton的Content值相同
    case "Ink Mode!":
      this.myInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
    break;

    case "Erase Mode!":
      this.myInkCanvas.EditingMode = InkCanvasEditingMode.EraseByStroke;
    break;

    case "Select Mode!":
      this.myInkCanvas.EditingMode = InkCanvasEditingMode.Select;
    break;
  }
}
```

此外，在窗口的构造函数中设置默认模式为Ink，并且设置ComboBox的默认选项（下一节将详细介绍该控件）：

```
public MainWindow()
{
  this.InitializeComponent();

  // 默认模式为Ink
  this.myInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
  this.inkRadio.IsChecked = true;
  this.comboColors.SelectedIndex=0;
}
```

再次按下F5键运行程序。使用Ink模式绘制一些数据，然后使用Erase模式移除前面输入的鼠标笔画（鼠标图标自动变为橡皮）。最后，使用Select模式，将鼠标以套索的方式选中一些笔画。圈中某项后，可以在画布中移动或调整其大小。图28-33显示了正在起作用的编辑模式。

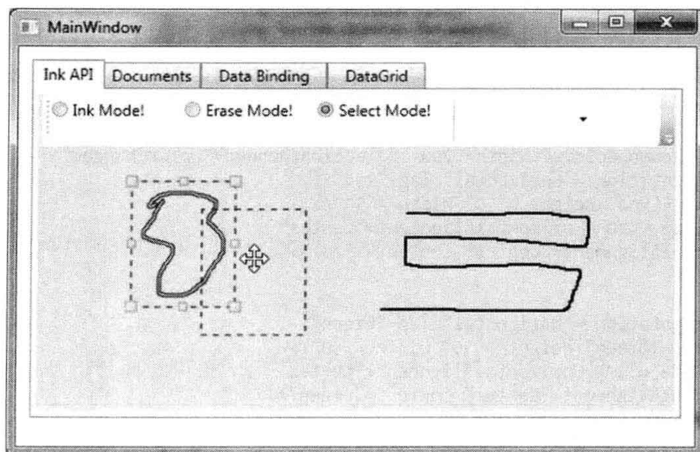


图28-33　InkCanvas编辑模式实战演练

28.7.5 ComboBox控件

有3种方法确定ComboBox控件（或ListBox控件）所选中的项。SelectedIndex属性可以得到被选项的数字索引（从0开始，-1表示没有选中项）。SelectedItem属性可以得到选中列表中的对象。SelectedValue可以得到被选对象的值（通常要调用ToString()方法）。

该选项卡还需要添加最后一些代码来改变在InkCanvas中输入的笔画颜色。InkCanvas的DefaultDrawingAttributes属性返回DrawingAttributes对象，可以用来配置各种笔尖，如大小、颜色（以及其他设置）。用下面的实现更新ColorChanged()方法：

```
private void ColorChanged(object sender,
  System.Windows.Controls.SelectionChangedEventArgs e)
{
  // 获取组合框中选中的值
  string colorToUse =
    (this.comboColors.SelectedItem as ComboBoxItem).Content.ToString();

  // 更改笔画呈现的颜色
  this.myInkCanvas.DefaultDrawingAttributes.Color =
    (Color)ColorConverter.ConvertFromString(colorToUse);
}
```

ComboBox包含ComboBoxItems的集合，生成的XAML的定义如下：

```
<ComboBox x:Name="comboColors" Width="100" SelectionChanged="ColorChanged">
  <ComboBoxItem Content="Red"/>
  <ComboBoxItem Content="Green"/>
  <ComboBoxItem Content="Blue"/>
</ComboBox>
```

在调用SelectedItem获取选中的ComboBoxItem时，得到的是通用的Object。将Object转换为ComboBoxItem后，才能得到Content的值，分别为字符串Red、Green、Blue。然后使用方便的ColorConverter工具类将string转换为Color对象。再次运行程序，就可以改变所呈现图像的颜色了。

注意，ComboBox和ListBox控件都可以包含复杂内容，而不仅仅限于文本数据的列表。要对这一点有个感性认识，可以打开窗口的XAML编辑器，更改ComboBox的定义，使其包含一组<StackPanel>元素，其中每个元素都包含一个<Ellipse>和一个<Label>（注意ComboBox的Width为200）：

```
<ComboBox x:Name="comboColors" Width="200" SelectionChanged="ColorChanged">
  <StackPanel Orientation ="Horizontal" Tag="Red">
    <Ellipse Fill ="Red" Height ="50" Width ="50"/>
    <Label FontSize ="20" HorizontalAlignment="Center"
           VerticalAlignment="Center" Content="Red"/>
  </StackPanel>

  <StackPanel Orientation ="Horizontal" Tag="Green">
    <Ellipse Fill ="Green" Height ="50" Width ="50"/>
    <Label FontSize ="20" HorizontalAlignment="Center"
           VerticalAlignment="Center" Content="Green"/>
  </StackPanel>

  <StackPanel Orientation ="Horizontal" Tag="Blue">
    <Ellipse Fill ="Blue" Height ="50" Width ="50"/>
```

```
    <Label FontSize ="20" HorizontalAlignment="Center"
           VerticalAlignment="Center" Content="Blue"/>
  </StackPanel>
</ComboBox>
```

注意，每个StackPanel都设置了Tag属性的值，这是一种获取用户选中面板（还有更好的方式，但现在这样就足够了）的简单、快捷、便利的方式。这样调整之后，需要更改ColorChanged()方法的实现，如下：

```
private void ColorChanged(object sender,
  System.Windows.Controls.SelectionChangedEventArgs e)
{
  // 获取所选中的StackPanel的Tag
  string colorToUse = (this.comboColors.SelectedItem
      as StackPanel).Tag.ToString();
  ...
}
```

现在再次运行程序，注意独特的组合框（如图28-34所示）。

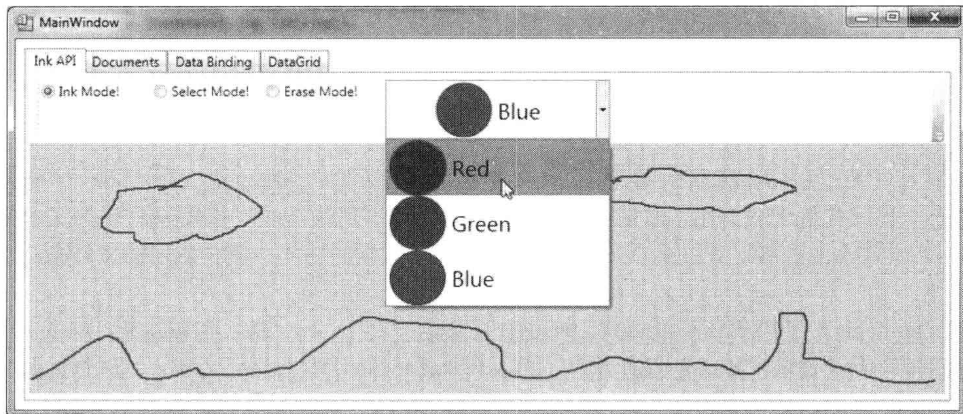


图28-34　自定义组合框，这归功于WPF内容模型

28.7.6　保存、加载和清除InkCanvas数据

这个选项卡的最后一部分是对画布数据进行保存、加载和清除。此时，你应该对设计UI很熟悉了，因此我们只进行简要地说明。

在代码文件中引入System.IO和System.Windows.Ink命名空间。在ToolBar中添加3个Button控件，分别命名为btnSave、btnLoad、btnClear。然后，处理各个控件的Click事件，并实现处理程序，如下：

```
private void SaveData(object sender, System.Windows.RoutedEventArgs e)
{
  // 在本地文件中保存InkCanvas的所有数据
  using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Create))
  {
    this.myInkCanvas.Strokes.Save(fs);
    fs.Close();
  }
```

```
}

private void LoadData(object sender, System.Windows.RoutedEventArgs e)
{
  // 从文件中获取数据，填充StrokeCollection
  using(FileStream fs = new FileStream("StrokeData.bin",
    FileMode.Open, FileAccess.Read))
  {
    StrokeCollection strokes = new StrokeCollection(fs);
    this.myInkCanvas.Strokes = strokes;
  }
}

private void Clear(object sender, System.Windows.RoutedEventArgs e)
{
  // 清除所有笔画
  this.myInkCanvas.Strokes.Clear();
}
```

现在我们可以将数据保存到文件中，从文件中加载数据以及清除InkCanvas中的所有数据。现在我们介绍完了TabControl中的第一个选项卡，并且学习了WPF数字墨水API。当然，对于该技术，还有内容没有涉及。但它会给你继续深入学习打下良好的基础。下面我们将学习如何使用WPF Documents API。

28.8 Documents API

WPF发布了很多可以获取或显示简单文本数据的控件，如Label、TextBox、TextBlock和PasswordBox。这些控件十分有用，但是某些WPF应用程序要使用复杂的、高度格式化的文本数据，类似Adobe PDF文件中的内容。WPF的Documents API提供了这样的功能，但它使用的文件格式是XPS而不是PDF。

利用Documents API中System.Windows.Documents命名空间下的一些类，可以构建用来打印的文档。该命名空间包含大量表示富XPS文档的类型，如List、Paragraph、Section、Table、LineBreak、Figure、Floater和Span。

28.8.1 块元素和内联元素

在形式上，添加到XPS文档中的项属于两大类型：块元素和内联元素。属于块元素的类都扩展了System.Windows.Documents.Block基类，例如List、Paragraph、BlockUIContainer、Section和Table。这些类可以将其他内容组合在一起（如列表包含段落数据，而段落包含不同文本格式的子段落）。

属于内联元素的类都扩展了System.Windows.Documents.Inline基类。可以将内联元素嵌入另一个块项目中（或嵌入到块元素中的内联元素中）。常见的内联元素有Run、Span、LineBreak、Figure和Floater。

使用专业编辑器构建富文档时，你可能会经常遇到这些类的名称。与其他WPF控件一样，你可以在XAML和代码中配置这些类。因此，你可以声明一个空的<Paragraph>，并在运行时进行填充（下例将演示如何这么做）。

28.8.2 文档布局管理器

你可能认为可以直接在Grid这样的面板容器中放置块元素和内联元素。然而实际上，你需要将它

们包裹在<FlowDocument>元素或<FixedDocument>元素中。

如果希望用户更改数据显示在计算机屏幕上的方式，将项目放置在FlowDocument中是很理想的。这样用户可以缩放文本或更改数据的显示方式（如显示为长页面，或分列显示）。而对于真正用于打印（所见即所得）的、不能更改的文档数据，则最好使用FixedDocument。

在本例中，我们只关注FlowDocument容器。向FlowDocument中插入内联项目或块项目时，FlowDocument对象将放置于4个XPS布局管理器中的一个，如表28-4所示。

表28-4 XPS控件布局管理器

面板控件	作 用
FlowDocumentReader	显示FlowDocument中的数据，并支持缩放、搜索和不同形式的内容布局
FlowDocumentScrollViewer	显示FlowDocument中的数据，但数据表现为带滚动条的单一文档。该容器不支持缩放、搜索或其他布局模式
RichTextBox	显示FlowDocument中的数据，并支持用户编辑
FlowDocumentPageViewer	逐页显示文档，每次显示一页。数据可以被缩放，但不能被检索

FlowDocumentReader管理器能以功能最丰富的方式显示FlowDocument。这时，用户可以改变布局，在文档中搜索单词，以及对提供的UI中的数据进行缩放。该容器（以及FlowDocumentScrollViewer和FlowDocumentPageViewer）的一个局限是，所显示的内容是只读的。如果希望允许用户在FlowDocument中输入新的信息，可以将其包裹在RichTextBox控件里。

28.9 构建 Documents 选项卡

单击TabItem中的Documents选项卡，使用设计器打开该控件进行编辑。现在TabItem的直接子元素为默认的<Grid>控件，使用Document Outline窗口将其修改为StackPanel。该选项卡将显示允许用户对选中文本进行突出显示的FlowDocument，以及使用Sticky Notes API添加批注。

按如下方式定义ToolBar控件，其中包含3个简单的（未命名的）Button控件。稍后我们将为这些控件装备一些新的命令，因此现在不必在代码中引用它们（随意直接进入XAML，如果喜欢也可以使用IDE）。

```
<TabItem x:Name="tabDocuments" Header="Documents" VerticalAlignment="Bottom"
        Height="20">
  <StackPanel>
    <ToolBar>
      <Button BorderBrush="Green" Content="Add Sticky Note"/>
      <Button BorderBrush="Green" Content="Delete Sticky Notes"/>
      <Button BorderBrush="Green" Content="Highlight Text"/>
    </ToolBar>
  </StackPanel>
</TabItem>
```

你也可以更新Visual Studio的Toolbox添加一个FlowDocumentReader控件（跟添加InkCanvas使用的技术相同），或使用XAML编辑器手动更新当前TabItem。

不管使用哪种方法，将FlowDocumentReader控件放置在StackPanel中，命名为myDocumentReader，并进行拉伸使其占据StackPanel的表面。为这个新组件添加一个空的<FlowDocument>。

```
<FlowDocumentReader x:Name="myDocumentReader" Height="269.4">
  <FlowDocument/>
</FlowDocumentReader>
```

此时，就可以向<FlowDocument>元素中添加文档类了（如List、Paragraph、Section、Table、LineBreak、Figure、Floater和Span）。下面是设置<FlowDocument>的一种方法。

```
<FlowDocumentReader x:Name="myDocumentReader" Height="269.4">
  <FlowDocument>
    <Section Foreground = "Yellow" Background = "Black">
      <Paragraph FontSize = "20">
        Here are some fun facts about the WPF Documents API!
      </Paragraph>
    </Section>
    <List/>
    <Paragraph/>
  </FlowDocument>
</FlowDocumentReader>
```

现在运行程序（按F5键），你已经可以缩放文档字体（使用右下角的滑动条）、搜索关键字（使用左下角的搜索编辑器）并以三种方式（使用布局按钮）显示数据。

继续学习之前，你可能希望编辑XAML以使用除了FlowDocumentReader以外的其他FlowDocument容器，如FlowDocumentScrollViewer或RichTextBox。修改之后，再次运行应用程序，并注意处理文档数据的不同之处。但要记得最后要改回FlowDocumentReader。

28.9.1　使用代码填充FlowDocument

现在，我们在代码中构建List块和Paragraph块。这是十分重要的，因为你也许需要基于用户输入、外部文件、数据库信息等来填充FlowDocument。首先用XAML编辑器为List和Paragraph元素起一个合适的名字，以便在代码中进行访问：

```
<List x:Name="listOfFunFacts"/>
<Paragraph x:Name="paraBodyText"/>
```

在代码文件中，定义一个新的私有方法PopulateDocument()。该方法先在List中添加一组ListItem，每个ListItem中都包含拥有单个Run的Paragraph。同样，该辅助方法还使用3个独立的Run对象动态构建格式化的段落，如下所示：

```
private void PopulateDocument()
{
  // 向List项中添加一些数据
  this.listOfFunFacts.FontSize = 14;
  this.listOfFunFacts.MarkerStyle = TextMarkerStyle.Circle;
  this.listOfFunFacts.ListItems.Add(new ListItem( new
    Paragraph(new Run("Fixed documents are for WYSIWYG print ready docs!"))));
  this.listOfFunFacts.ListItems.Add(new ListItem(
    new Paragraph(new Run("The API supports tables and embedded figures!"))));
  this.listOfFunFacts.ListItems.Add(new ListItem(
    new Paragraph(new Run("Flow documents are read only!"))));
  this.listOfFunFacts.ListItems.Add(new ListItem(new Paragraph(new Run
    ("BlockUIContainer allows you to embed WPF controls in the document!")
    )));

  // 向Paragraph中添加一些数据
```

```
    // 句子的第一部分
    Run prefix = new Run("This paragraph was generated ");

    // 段落的中间部分
    Bold b = new Bold();
    Run infix = new Run("dynamically");
    infix.Foreground = Brushes.Red;
    infix.FontSize = 30;
    b.Inlines.Add(infix);

    // 段落的最后部分
    Run suffix = new Run(" at runtime!");

    // 向内联元素的集合中添加段落的各个部分
    this.paraBodyText.Inlines.Add(prefix);
    this.paraBodyText.Inlines.Add(infix);
    this.paraBodyText.Inlines.Add(suffix);
}
```

在窗口的构造函数中调用该方法，然后运行程序，可以看到新的动态生成的文档内容。

28.9.2 启用批注和便签

到目前为止，一切都很顺利。你可以使用XAML和C#代码构建包含数据的文档。但我们仍然需要Documents选项卡工具条中的3个按钮。WPF发布了一系列专门用于Documents API的命令，可以允许用户选择文档内容或添加便签批注。最棒的是，添加所有这些功能只需要很少的代码（和标记）。

用于Documents API的命令对象位于PresentationFramework.dll的System.Windows.Annotations命名空间下。因此，我们需要在<Window>的开放式元素中定义自定义的XML命名空间，才能在XAML中使用这些对象（前缀为a）：

```
<Window
...
  xmlns:a=
  "clr-namespace:System.Windows.Annotations;assembly=PresentationFramework"
  x:Class="WpfControlsAndAPIs.MainWindow"
  x:Name="Window"
  Title="MainWindow"
  Width="856" Height="383" mc:Ignorable="d"
    WindowStartupLocation="CenterScreen" >
...
</Window>
```

现在更新3个<Button>的定义，用3个批注命令设置其Command属性：

```
<ToolBar>
  <Button BorderBrush="Green" Content="Add Sticky Note"
          Command="a:AnnotationService.CreateTextStickyNoteCommand"/>
  <Button BorderBrush="Green" Content="Delete Sticky Notes"
          Command="a:AnnotationService.DeleteStickyNotesCommand"/>
  <Button BorderBrush="Green" Content="Highlight Text"
          Command="a:AnnotationService.CreateHighlightCommand"/>
</ToolBar>
```

最后要做的事情是启用FlowDocumentReader对象（即myDocumentReader）的批注服务。在类中添加另一个私有方法EnableAnnotations()，并在窗口的构造函数中调用它。现在引入如下的命名空间：

```
using System.Windows.Annotations;
using System.Windows.Annotations.Storage;
```

然后实现以下方法：

```
private void EnableAnnotations()
{
  // 创建用于FlowDocumentReader的AnnotationService对象
  AnnotationService anoService = new AnnotationService(myDocumentReader);

  // 创建用于存放批注的MemoryStream
  MemoryStream anoStream = new MemoryStream();

  // 根据MemoryStream创建基于XML的存储
  // 可以使用这个对象以编程的方式添加、删除或查找批注
  AnnotationStore store = new XmlStreamStore(anoStream);

  // 启用批注服务
  anoService.Enable(store);
}
```

AnnotationService类可以使给定的文档布局管理器支持批注。在调用该对象的Enable()方法之前，需要为该对象提供一个保存批注数据的位置，本例使用由MemoryStream对象表示的一块内存区域。注意，我们使用AnnotationStore将AnnotationService和Stream连接在一起。

现在运行应用程序。选取一些文本，然后单击Add Sticky Note按钮，并输入一些信息。同样选取一些文本，还可以突出显示数据（颜色默认为黄色）。最后，选中便签，单击Delete Sticky Note按钮，可以进行删除，如图28-35所示。

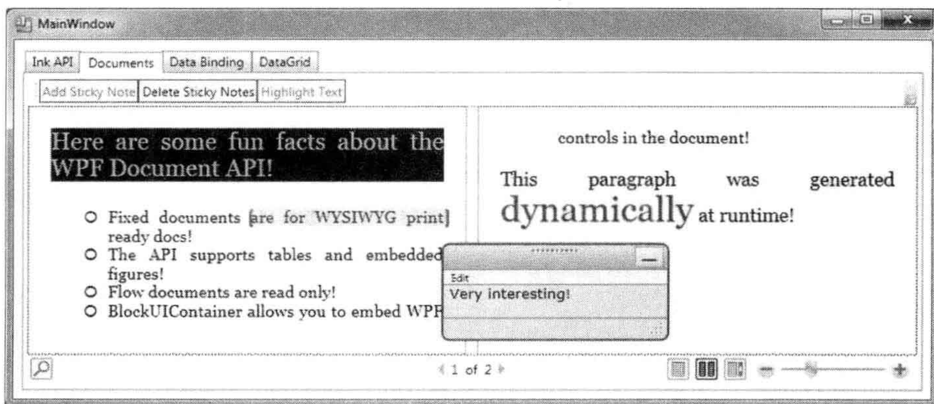


图28-35　便签

28.9.3　保存和加载流文档

在Documents API的最后，我们来看看将文档保存到文件以及从文件中读取文档是多么简单。记住，只有在RichTextBox中显示FlowDocument对象时，用户才能编辑文档。但你完全可以在运行时动态创建部分文档，因此可能希望将其保存以便之后使用。加载XPS文档的功能在很多WPF应用程序中都是十分有用的，因为你可能希望在运行时定义空文档并加载它。

下面的代码片段在Documents选项卡的工具条中添加了两个Button，如下（注意，没有在标记中处理任何事件）：

```
<Button x:Name="btnSaveDoc" HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch" Width="75" Content="Save Doc"/>
<Button x:Name="btnLoadDoc" HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch" Width="75" Content="Load Doc"/>
```

在窗口的构造函数中，编写如下的Lambda表达式，来保存和加载FlowDocument数据（需要引入System.Windows.Markup命名空间来访问XamlReader和XamlWriter类）：

```
public MainWindow()
{
...
  // 为保存和加载流文档装配Click事件处理程序
  btnSaveDoc.Click += (o, s) =>
  {
    using(FileStream fStream = File.Open(
      "documentData.xaml", FileMode.Create))
    {
      XamlWriter.Save(this.myDocumentReader.Document, fStream);
    }
  };

  btnLoadDoc.Click += (o, s) =>
  {
    using(FileStream fStream = File.Open("documentData.xaml", FileMode.Open))
    {
      try
      {
        FlowDocument doc = XamlReader.Load(fStream) as FlowDocument;
        this.myDocumentReader.Document = doc;
      }
      catch(Exception ex) {MessageBox.Show(ex.Message, "Error Loading Doc!");}
    }
  };
}
```

这就是保存文档时我们要做的全部工作（注意我们并没有保存任何批注，你可以使用批注服务完成这项任务）。如果单击Save按钮，将在\bin\Debug文件夹下看到新的*.xaml文件。该文件包含当前的文档数据。

至此对于WPF Documents API的介绍就告一段落了。当然，对此我们还有很多方面没有涉及，但你应该已经了解了大量的基础知识。在本章最后，我们将学习一些关于数据绑定的内容并完成当前的应用程序。

28.10 WPF 数据绑定模型

各种数据绑定操作的目标通常是控件。简单地说，*数据绑定*是将应用程序生命周期过程中可能会改变的数据值连接到控件属性的行为。这样可以在代码中使用户界面元素显示变量的状态。例如，可以使用数据绑定完成以下任务：

- 根据给定对象的某个Boolean属性检查CheckBox控件；
- 在DataGrid对象中显示来自关系数据库表中的数据；

- 将Label连接到表示文件夹中文件数量的整数。

在使用固有的WPF数据绑定引擎时，必须注意绑定操作中源和目标的区别。数据绑定操作中的源是指数据本身（如Boolean属性或相关的数据），而目标是指使用数据内容的UI控件属性（如CheckBox或TextBox）。

实际上，WPF数据绑定基础结构的使用并不是强制的。但如果要自己得到数据绑定逻辑，源和目标之间的连接常常会需要处理各种事件并编写程序代码。例如，如果窗口中含有一个ScrollBar，需要将其值显示在一个Label类型上，你就需要处理ScrollBar的ValueChanged事件，并相应地更新Label的内容。

尽管如此，你可以使用WPF数据绑定直接在XAML中（或在代码文件中使用C#代码）连接源和目标，而不需要处理各种事件和硬编码。同样，根据数据绑定逻辑的设置，可以使源和目标在任何一方的值有变化时都能保持同步。

28.10.1　构建Data Binding选项卡

使用Document Outline编辑器，将第三个选项卡的Grid改为StackPanel。然后使用Toolbox和Visual Studio的Properties编辑器构建如下的初始布局：

```
<TabItem x:Name="tabDataBinding" Header="Data Binding">
  <StackPanel Width="250">
    <Label Content="Move the scroll bar to see the current value"/>

    <!-- 滚动条的值是该数据绑定的数据源 -->
    <ScrollBar x:Name="mySB" Orientation="Horizontal" Height="30"
           Minimum = "1" Maximum = "100" LargeChange="1" SmallChange="1"/>

    <!-- Label的内容将绑定到滚动条上-->
    <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
           BorderThickness="2" Content = "0"/>
  </StackPanel>
</TabItem>
```

注意，<ScrollBar>对象（mySB）的值在1到100之间。我们的目的是确保在移动滚动条（或单击左右箭头）时，Label能自动更新当前的值。目前，Label控件的Content属性的值设为"0",我们可以通过数据绑定操作改变这个值。

28.10.2　使用Visual Studio建立数据绑定

在XAML中定义绑定的方法是使用{Binding}标记扩展。如果用Visual Studio在控件之间建立绑定，就更为简单。例如，找到Label对象的Content属性（位于Properties窗口的Common区域），单击旁边非常小的白色正方形，打开快捷菜单，选择Data Binding（如图28-36所示）。

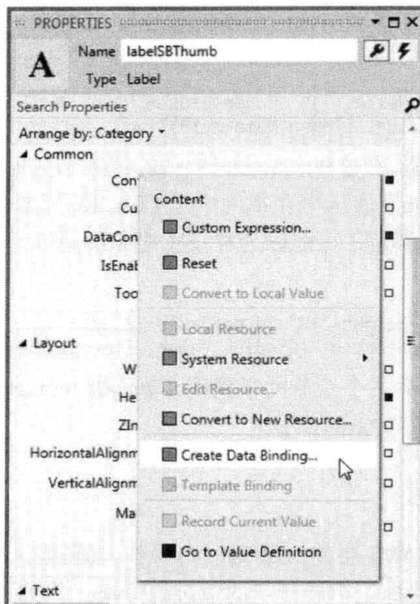


图28-36 配置数据绑定操作

从Binding Type下拉列表中选择ElementName选项，这会给出你的XAML文件中所有的项目，我们可以将它们作为数据绑定操作的源。在元素名称树控件中找到`ScrollBar`对象（`mySB`）。在Path树中找到`Value`属性（如图28-37所示）。完成之后单击OK按钮。

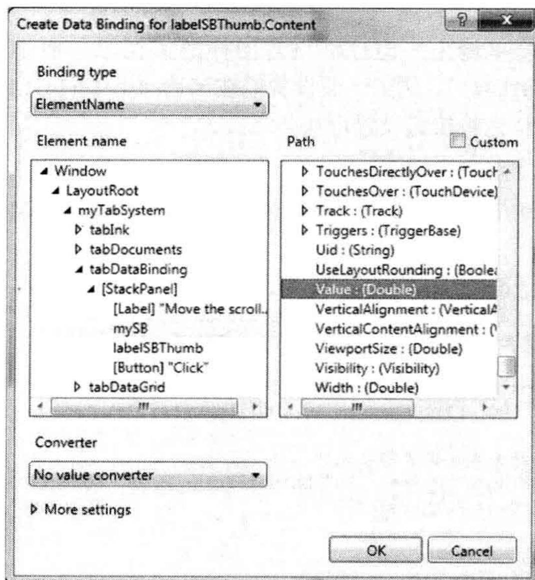


图28-37 选择源对象及其属性

再次运行该程序，可以看到在移动滑块时标签的值也随之更新。现在我们来看看数据绑定工具为我们生成的XAML：

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
       Content = "{Binding Value, ElementName=mySB}"/>
```

注意Label的Content属性的值。在这里，ElementName的值表示数据绑定操作的源（ScrollBar对象），而Binding关键字后面的项（Value）表示（在本例中）要获取的元素的属性。

如果你以前使用过WPF数据绑定，可能以为在设置对象的属性时必须使用Path标记。例如，下面的标记同样可以更新Label：

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2" Content = "{Binding Value, ElementName=mySB}"/>
```

默认情况下，如果属性不是另一个对象的子属性（如myObject.MyProperty.Object2.Property2），Visual Studio将省略数据绑定操作的Path=部分。

28.10.3 DataContext属性

你还可以使用其他格式在XAML中定义数据绑定操作。通过显式地将数据绑定操作的源设置给DataContext，可以将由{Binding}标记扩展指定的值进行分离。例如：

```
<!-- 用DataContext将对象和值分开 -->
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2"
       DataContext = "{Binding ElementName=mySB}"
       Content = "{Binding Path=Value}" />
```

对于当前示例来说，如果这样修改标记，将得到同样的输出结果。那么什么时候需要显式设置DataContext属性呢？由于子元素可以继承源在标记树中的值，因此是十分有帮助的。

使用这种方法，可以简单地为一组控件设置同样的数据源，而不需要为多个控件重复设置"{Binding ElementName=X, Path=Y}"。例如，假设我们在该选项卡中的<StackPanel>中添加了一个新的Button（稍后你就会知道为什么将按钮设置得如此之大）：

```
<Button Content="Click" Height="140"/>
```

你可以使用Visual Studio为多个控件生成数据绑定，也可以使用XAML编辑器手工输入修改后的标记：

```
<!-- 注意StackPanel设置了DataContext属性 -->
<StackPanel Width="250" DataContext = "{Binding ElementName=mySB}">
  <Label Content="Move the scroll bar to see the current value"/>

  <ScrollBar Orientation="Horizontal" Height="30" Name="mySB"
             Maximum = "100" LargeChange="1" SmallChange="1"/>

  <!-- 两个UI元素以独特的方式使用滚动条的值 -->
  <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
         Content = "{Binding Path=Value}"/>

  <Button Content="Click" Height="200"
          FontSize = "{Binding Path=Value}"/>
</StackPanel>
```

我们在这里直接设置了<StackPanel>的DataContext属性。因此，在移动滑块时，不仅Label的值会发生改变，Button中的字体也会相应地放大或缩小（图28-38显示了可能的输出结果）。

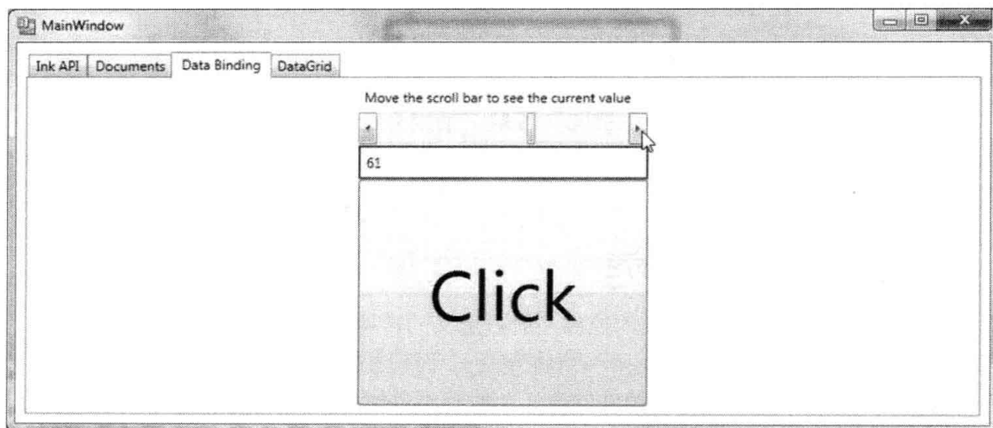


图28-38 将ScrollBar的值绑定到Label和Button

28.10.4 使用IValueConverter进行数据转换

ScrollBar类型使用double而不是整数来表示滑块的值。因此在拖动滑块时，Label中将显示不同的浮点数（如61.0576923076923）。这对于用户来说是非常不直观的，因为他们更希望看到整数（如61、62和63）。

如果希望将数据绑定操作的值转换为其他格式，可以创建一个自定义类，实现System.Windows.Data命名空间中的IValueConverter接口。该接口定义了两个成员，可以将值转换为目标类型，或从目标类型转换为值（双向数据绑定）。然后它可以用来进一步限定数据绑定操作的过程。

假设我们要在Label控件中显示整数，可以构建如下的自定义转换类。激活Project→Add Class菜单，插入名为MyDoubleConverter的类。然后添加如下代码：

```
class MyDoubleConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
                        System.Globalization.CultureInfo culture)
  {
    // 将double转换为int
    double v = (double)value;
    return (int)v;
  }

  public object ConvertBack(object value, Type targetType, object parameter,
                            System.Globalization.CultureInfo culture)
  {
    // 这里我们不用担心“双向”绑定，因此直接返回value
    return value;
  }
}
```

28

当值从源（ScrollBar）转换为目标（TextBox的Text属性）时，会调用Convert()方法。它接收很多参数，但在该示例中我们只操作传入的代表当前double值的object参数。我们使用该类型将其转换为整型，并返回。

当值从目标传递回源时（需要启用双向绑定模式），将调用ConvertBack()方法。这里我们直接返回value。这样如果在TextBox中输入浮点值（如99.9），而当用户将焦点移开该控件时，将自动转换为整数值（如99）。之所以会发生这种“免费的”转换，是因为在调用了ConvertBack()之后又再次调用了Convert()。如果在ConvertBack()中简单地返回null，文本框将仍然显示浮点数，这样绑定就不同步了。

28.10.5 在代码中建立数据绑定

有了这个类，就可以向要使用该类的控件注册自定义的转换器。仅在XAML中就可以完成该任务，但这需要定义一些自定义的对象资源（下一章将介绍）。而现在，我们可以在代码中注册数据转换类。先清理一下数据绑定选项卡中<Label>控件的定义，使其不再使用{Binding}标记扩展：

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2" Content = "0"/>
```

在窗口的构造函数中，调用一个新的私有辅助函数SetBindings()。在该方法中添加如下代码：

```
private void SetBindings()
{
  // 创建Binding对象
  Binding b = new Binding();

  // 注册转换器、源和路径
  b.Converter = new MyDoubleConverter();
  b.Source = this.mySB;
  b.Path = new PropertyPath("Value");

  // 调用Label的SetBinding()方法
  this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

该函数中唯一有点陌生的大概就是SetBinding()方法。注意它的第一个参数调用了Label类中的静态只读字段ContentProperty。现在我们所指定的叫做*依赖属性*，这将在第31章中进行介绍。此时，只需要知道在代码中设置绑定时，第一个参数总是需要指定进行绑定的类的名称（本例中为Label），然后再调用包含-Property后缀的实际属性（同样将在第31章中介绍）。无论如何，运行程序将看到只打印整数的Label。

28.10.6 构建DataGrid选项卡

前面的示例演示了如何为一个数据绑定操作配置两个（或多个）控件。我们还可能从XML文件、数据库数据和内存对象中绑定数据。为了完成该示例，我们来设计选项卡控件中的最后一个选项卡，它将显示从AutoLot数据库的Inventory表中获取的数据。

与其他选项卡一样，我们先将当前的Grid更改为StackPanel。我们使用Visual Studio直接在XAML中进行修改。然后在新的StackPanel中定义一个DataGrid控件，命名为gridInventory：

```
<TabItem x:Name="tabDataGrid" Header="DataGrid">
  <StackPanel>
    <DataGrid x:Name="gridInventory" Height="288"/>
  </StackPanel>
</TabItem>
```

接下来，引用第23章中创建的AutoLotDAL.dll程序集(使用了Entity Framework)。此时，Visual Studio会自动引用System.Data. Entity.dll，以及相关的App.config文件，该文件包含了所需的连接字符串数据（检查Solution Explorer确认情况，如果没有，手动添加必需的项目）。

打开窗口的代码文件，添加最后一个辅助函数ConfigureGrid()，并在构造函数中调用它。引入AutoLotDAL命名空间，我们所要做的仅仅是添加这几行代码：

```
private void ConfigureGrid()
{
  using (AutoLotEntities context = new AutoLotEntities())
  {
    // 构建从Inventory表中获取数据的LINQ查询
    var dataToShow = from c in context.Inventories
      select new { c.CarID, c.Make, c.Color, c.PetName };
    this.gridInventory.ItemsSource = dataToShow;
  }
}
```

注意，我们没有直接将context.Inventories绑定到网格的ItemSource集合，而是对其构建了一个貌似请求同一个数据的LINQ查询。这么做的原因是，Inventory对象集还包含其他未映射到物理数据库的EF属性，如果不这么做，它们也将显示在网格中。

如果这时运行项目，会看到一个极其普通的网格。为了避免视觉疲劳，我们使用Visual Studio的Properties窗口编辑DataGrid的Rows属性，将AlternationCount属性的值至少设置为2，并使用AlternatingRowBackground和RowBackground属性集成的编辑器选择一个自定义画刷，该示例的最后一个选项卡如图28-39所示。

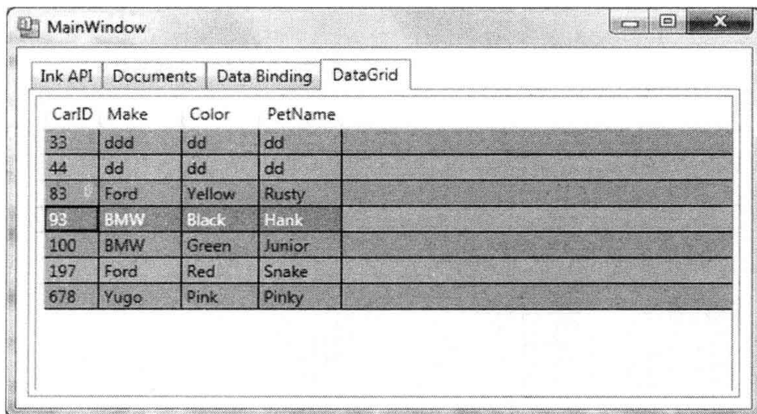


图28-39 项目的最后一个选项卡

该示例就介绍到这了。在接下来的几章中，我们将使用一些其他的控件。现在，你应该对使用Visual Studio以及手工编写XAML和C#代码来构建UI的过程，有了一定的认识。

源代码 WpfControlsAndAPIs项目的源代码位于Chapter 28子目录下。

28.11 小结

本章开头概述了控件工具包和布局管理器（面板），随后介绍了一些WPF控件。第一个示例构建了一个简单的文字处理应用程序，演示了WPF集成的拼写检查功能，以及如何使用菜单系统、状态条和工具条构建主窗口。

更重要的是，我们学习了如何使用WPF命令。可以将这些控件无关的事件附加到UI元素或某种输入方式，来自动集成方便的服务（如剪贴板操作）。

我们还学习了通过集成的可视设计器使用Visual Studio构建用户界面，特别是使用各种工具构建复杂的用户界面，并同时介绍了WPF Ink和Document API。我们还介绍了WPF数据绑定操作，如如何使用WPF DataGrid类显示自定义的AutoLot数据库中的数据。

第 29 章

WPF图形呈现服务

本章将介绍WPF的图形呈现能力。你将看到WPF提供了三种不同的方式来呈现图形数据——形状（shape）、绘图（drawing）及可视化（visual）。在理解了每种方法的利弊之后，我们将开始学习使用System.Windows.Shapes下的类来与2D图形世界进行交互。之后，你将看到如何使用绘图和几何图形（geometry）以一种更轻量级的方式来呈现2D数据。最后，你将学习如何使用可视化层提供最强大的功能和最优的性能。

随着学习的深入，我们还将探讨一些相关的话题，例如创建自定义画刷（brush）和画笔（pen）、在呈现时使用图形变换（graphical transformation）以及执行命中测试（hit-test）操作。你还将看到Visual Studio的集成工具和新的Expression Design工具，以及它们是如何简化图形编码的。

说明 图形（graphics）是WPF开发的关键部分。即使我们构建的不是偏重图形的应用程序（如视频游戏或多媒体应用程序），当你使用控件模板、动画和数据绑定定制等服务的时候，也应该了解本章讨论的话题。

29.1 理解 WPF 的图形呈现服务

WPF使用一种特殊的图形呈现方式，叫做保留模式图形（retained mode graphics）。简言之，当我们使用XAML或程序代码来呈现图形时，将这些可见元素持久化，保证它们被正确地重绘并以最优的方式刷新都是WPF的职责。这样当你呈现图形数据时，无论最终用户通过改变窗口大小隐藏图像、最小化窗口还是用一个窗口覆盖另一个，它总是存在的。

与此大相径庭的是，过去的微软图形呈现API（包括Windows Form的GDI+）都是即时模式（immediate-mode）图形系统。在这种模式中，将由程序员来保证呈现的可视元素被正确地“记忆”，并在应用程序生命周期正确地更新。举个例子，在Windows Form应用程序中呈现一个矩形包括几个部分：处理Paint事件（或重写OnPaint()虚方法），获取一个呈现矩形的Graphics对象，最重要的是，增加基础设施来保证当用户更改窗口大小时能够对图像进行持久化（例如，通过程序调用Invalidate()，创建表示矩形位置的成员变量）。

这种从即时模式到保留模式的观念转变，减少了程序员需要编写和维护的乱糟糟的图形代码。然而，我们并不是说WPF图形API与之前的呈现工具包完全不同。举例来说，WPF像GDI+一样支持多种画刷和画笔类型、命中测试、剪裁区域（clipping region）、图形变换，等等。因此，如果你有在GDI+

（或基于GDI的C/C++）的编程经验，就应该知道许多在WPF下进行基本呈现的知识。

WPF图形呈现方式

和WPF开发的其他方面一样，除了采用XAML或C#程序代码以外，还可以选择很多方式来实现图形呈现。具体地说，WPF提供了3种不同的方式来呈现图形数据。

- **形状（Shape）**：WPF提供了System.Windows.Shapes命名空间，定义了少量的类来呈现2D几何图形对象（如矩形、椭圆形、多边形等）。尽管这些类简单易用且功能强大，但如果滥用却会导致大量的内存占用。
- **绘图和几何图形（Drawing和Geometrie）**：WPF API提供的另一种呈现图形数据的方法是派生System.Windows.Media.Drawing抽象类。使用GeometryDrawing或ImageDrawing这样的类（和若干几何对象），你可以用一种轻量级（但功能有限）的方式来呈现图形数据。
- **可视化（Visual）**：在WPF中呈现图形数据最快速、最轻量级的方式就是使用可视化层，你只能通过C#代码来访问该层。使用System.Windows.Media.Visual的派生类可以直接与WPF图形子系统进行对话。

为相同的功能（如呈现图形数据）提供不同的实现方式，其目的是为了处理内存使用和优化应用程序性能。由于WPF是一个图形密集型系统，在一个应用程序的窗体表面呈现成百上千张图片并不是没有可能，因此所选择的实现方式（形状、绘图、可视化）将会给程序带来巨大的影响。

在构建WPF应用程序时，不同的情况要选用不同的方式。一般来说，如果你需要一些可以由用户操作（接收鼠标输入、显示工具提示等）的交互图形数据，应该使用System.Windows.Shapes命名空间下的那些成员。

相反，当你希望用XAML或C#展示复杂的、非交互的、基于矢量的图形数据时，绘图和几何图形更适合一些。尽管它们也能响应鼠标事件、命中测试和拖曳操作，但这需要编写更多的代码。

最后，如果需要快速地呈现大量图形数据，使用可视化层是最好的方法。假设你正在使用WPF构建一个有关科学的应用程序，可以描绘成千上万的数据点。使用可视化层能以最佳的方式呈现这些点。你将在本章后面看到，可视化层只能用C#代码访问，而不能使用XAML。

不管使用什么方式（形状、绘图和几何图形、可视化），你都会使用通用的图形基元，如画刷（用来填充图形内部）、画笔（用来绘制图形边框）以及变换对象（用来变换数据）。我们先从System.Windows.Shapes入手，来开启WPF图形呈现之旅。

说明　WPF还发布了用于呈现和操作3D图形的API，不过本书将不对此进行讨论。如果你希望在应用程序中使用3D图形，请参考.NET Framework 4.5 SDK文档。

29.2　使用形状呈现图形数据

System.Windows.Shapes命名空间的成员提供了简单易用、交互性强但同时也会占用大量内存的方式来呈现二维图像。该命名空间（定义在PresentationFramework.dll程序集中）非常小，只包含6个扩展了抽象类Shape的密封类：Ellipse、Rectangle、Line、Polygon、Polyline和Path。

我们新建一个名为RenderingWithShapes的WPF应用程序。现在，在Visual Studio中的对象浏览器里找到Shape类（如图29-1所示），并打开所有父节点，你将看到Shape的派生类从继承链中继承了大量的功能。

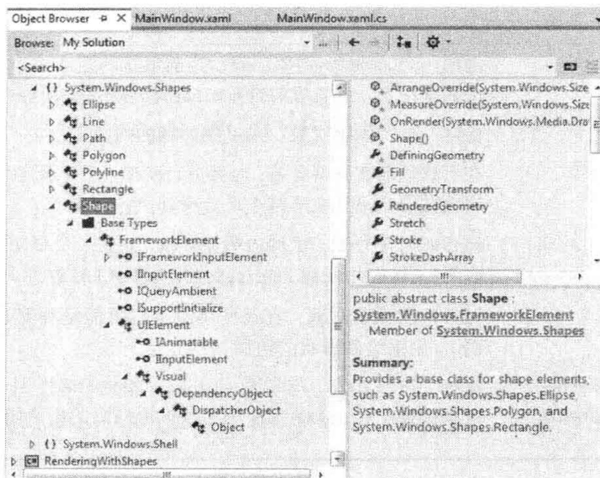


图29-1 Shape基类从父类中继承了大量的功能

这些父类也许会帮你回忆起前两章所学的内容。例如，UIElement定义了大量的方法来接收鼠标输入和处理拖曳事件，FrameworkElement定义了一些成员来处理尺寸、工具提示、鼠标光标，等等。在这种继承链下，当我们用Shape的派生类来呈现图形数据时，这些对象（就其交互性而言）仅仅是一个WPF控件。

例如，判断用户是否单击了所呈现的图形，其实就是在处理MouseDown事件。举例来说，如果你为一个Rectangle对象在初始的Window中的Grid里编写了如下所示的XAML：

```
<Rectangle x:Name="myRect" Height="30" Width="30"
          Fill="Green" MouseDown="myRect_MouseDown"/>
```


那么你可以在C#代码中实现MouseDown事件处理程序，在单击矩形时改变其背景颜色，如下所示：

```
private void myRect_MouseDown(object sender, MouseButtonEventArgs e)
{
  // 在单击时改变矩形的颜色
  myRect.Fill = Brushes.Pink;
}
```

与其他图形工具不同，将鼠标坐标手工映射到几何图形上、手工计算命中测试、呈现到离屏缓冲区等工作都不必再编写大量的基础代码。System.Windows.Shapes中的成员可以简单地响应你注册的事件，就像处理典型的WPF控件（如Button）一样。

形状的这些方便的功能也有一个缺点，它们会占用大量的内存。就像之前提到的，如果要构建一个会在屏幕上绘制上千个点的科学应用程序，使用形状将是一个错误的选择（基本上和呈现上千个Button对象所消耗的内存差不多）。但是，如果你只需要生成一个交互式的2D矢量图形，形状将是一个完美的选择。

除了从父类UIElement和FrameworkElement继承的功能之外，Shape还为每个派生类定义了许多成员。表29-1列出了一些常用的成员。

表29-1　Shape基类的主要属性

属　性	含　义
DefiningGeometry	返回一个Geometry对象，表示当前形状的整体尺寸。该对象仅包含用于呈现数据的坐标点，并且没有继承UIElement或FrameworkElement提供的功能
Fill	指定一个“画刷对象”来呈现形状的内部
GeometryTransform	在形状呈现在屏幕之前，对其进行变换。而继承自UIElement的RenderTransform属性在形状呈现在屏幕上之后对其执行变换
Stretch	描述如何在分配的区域内填充形状，如在一个布局管理器中的位置。这是用 相应的System.Windows.Media.Stretch枚举来控制的
Stroke	定义一个画刷对象，在某些情况下也可能是画笔对象（其实是一个变相的画刷），用来绘制形状的轮廓
StrokeDashArray，StrokeEnd-LineCap，StrokeStartLineCap，StrokeThickness	这些（以及其他）与笔画（stroke）相关的属性用于控制在呈现形状轮廓时的线条设置。在大多数情况下，这些属性可以设定用来绘制轮廓或线条的画刷

说明　如果你忘记了设置Fill和Stroke属性，WPF将使用“隐藏的”画刷，从而使形状在屏幕中不可见。

29.2.1　在画布中添加矩形、椭圆形和线条

本章后面的小节将会介绍使用Expression Design来生成图形数据的XAML描述信息。而现在，我们将使用XAML和C#来构建一个可以呈现形状的WPF应用程序，并在这个过程中学习一些命中测试的内容。首先，移除当前的Rectangle描述和C#事件处理程序逻辑。然后，更新<Window>中初始的XAML，定义一个包含<ToolBar>和<Canvas>的<DockPanel>。注意我们已经通过Name属性为每个项都指定了恰当的名称。

```
<DockPanel LastChildFill="True">
  <ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  </ToolBar>
  <Canvas Background="LightBlue" Name="canvasDrawingArea"/>
</DockPanel>
```

现在用一组<RadioButton>对象来填充<ToolBar>，每个<RadioButton>的内容都是一个特定的Shape派生类。注意每个<RadioButton>的GroupName都是相同的（用来确保它们是互斥的），并且都赋予了恰当的名称：

```
<ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  <RadioButton Name="circleOption" GroupName="shapeSelection">
    <Ellipse Fill="Green" Height="35" Width="35" />
  </RadioButton>

  <RadioButton Name="rectOption" GroupName="shapeSelection">
```

```
    <Rectangle Fill="Red" Height="35"
               Width="35" RadiusY="10" RadiusX="10" />
  </RadioButton>

  <RadioButton Name="lineOption" GroupName="shapeSelection">
    <Line  Height="35" Width="35"
           StrokeThickness="10" Stroke="Blue"
           X1="10" Y1="10" Y2="25" X2="25"
           StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />
  </RadioButton>
</ToolBar>
```

如你所见，在XAML中定义Rectangle、Ellipse和Line对象十分简单并且不需要注释。Fill属性用来指定绘制形状内部的画刷。如果需要单色画刷，你只需要用硬编码的字符串指定一个颜色值，隐式类型转换就会生成正确的对象。有趣的是，Rectangle类型还定义了RadiusX和RadiusY属性，用来呈现圆角。

Line使用X1、X2、Y1、Y2属性来表示线条的起始和结束点坐标（因为高度和宽度对线条来说没什么意义）。在这里，我们建立了一些额外的属性来呈现Line的起始和结束点，以及如何设置笔画。图29-2显示了在Visual Studio WPF设计器中所呈现的工具条。

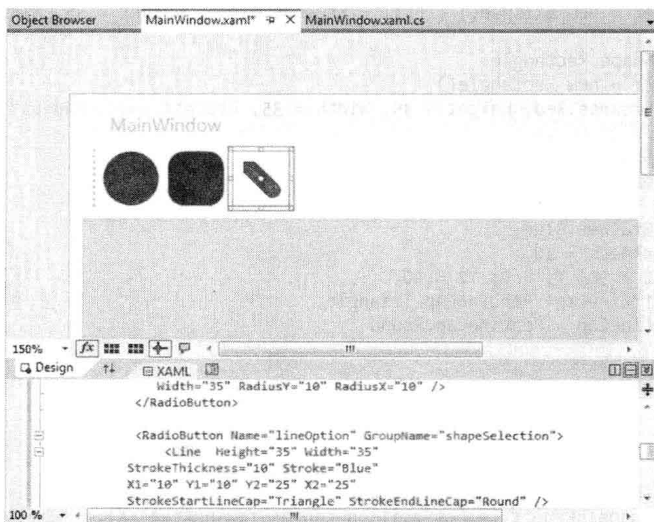


图29-2 用Shape作为RadioButton的内容

现在，使用Visual Studio中的Properties窗口，处理Canvas的MouseLeftButtonDown事件和每个RadioButton的Click事件。在C#代码中，我们的目标是在用户单击Canvas时呈现所选择的形状（圆形、正方形或线条）。首先，在Window派生类中定义一个内嵌的枚举（以及相应的成员变量）：

```
public partial class MainWindow : Window
{
  private enum SelectedShape
  { Circle, Rectangle, Line }

  private SelectedShape currentShape;
...
}
```

在每个Click事件处理程序中，将currentShape成员变量设置为正确的SelectedShape值。例如，如下所示的circleOption RadioButton的Click事件处理程序的实现代码。其他两个Click处理程序的代码与之类似：

```
private void circleOption_Click(object sender, RoutedEventArgs e)
{
  currentShape = SelectedShape.Circle;
}
```

在Canvas的MouseLeftButtonDown事件处理程序中，我们使用预先定义的尺寸来呈现正确的形状，用鼠标光标的X、Y位置作为起始点。下面是完整的代码和分析说明：

```
private void canvasDrawingArea_MouseLeftButtonDown(object sender,MouseButtonEventArgs e)
{
  Shape shapeToRender = null;

  // 设置要绘制的形状
  switch (currentShape)
  {
    case SelectedShape.Circle:
      shapeToRender = new Ellipse() { Fill = Brushes.Green, Height = 35, Width = 35 };
      break;
    case SelectedShape.Rectangle:
      shapeToRender = new Rectangle()
        { Fill = Brushes.Red, Height = 35, Width = 35, RadiusX = 10, RadiusY = 10 };
      break;
    case SelectedShape.Line:
      shapeToRender = new Line()
      {
        Stroke = Brushes.Blue,
        StrokeThickness = 10,
        X1 = 0, X2 = 50, Y1 = 0, Y2 = 50,
        StrokeStartLineCap= PenLineCap.Triangle,
        StrokeEndLineCap = PenLineCap.Round
      };
      break;
    default:
      return;
  }

  // 设置形状在画布中距离顶部和左侧的相对位置
  Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
  Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);
  // 绘制形状
  canvasDrawingArea.Children.Add(shapeToRender);
}
```

说明　你会发现在该方法中创建的Ellipse、Rectangle和Line对象与XAML中定义的对象有相同的属性值。你可能希望精简代码，但这需要理解WPF的对象资源，我们将在第30章中介绍。

如你所见，我们使用currentShape成员变量来创建正确的Shape派生类。然后使用传入的MouseButtonEventArgs分别设置距离Canvas边界顶部和左侧的值。最后也是最重要的，我们将Shape派生类添加到Canvas中声明的UIElement对象集合中。如果现在运行程序，你可以在画布中的任何位置进行单击，同时所选择的形状也将呈现在鼠标左键单击的位置。

29.2.2 在画布中移除矩形、圆形和线条

在包含一个对象集合的Canvas中，你可能不知道如何动态地移除一个对象，你也许会希望通过右击一个形状即可。你当然可以这么做，只需要使用System.Windows.Media命名空间下的VisualTreeHelper类就可以实现这个目的。在第31章中，我们将详细介绍“视觉树”和“逻辑树”的作用。在此之前，可以像下面的代码这样实现Canvas对象的MouseRightButtonDown事件处理程序：

```
private void canvasDrawingArea_MouseRightButtonDown(object sender, MouseButtonEventArgs e)
{
  // 首先，得到用户单击的X，Y 位置
  Point pt = e.GetPosition((Canvas)sender);
  // 使用VisualTreeHelper的HitTest()方法来检查用户是否单击了画布中的某个对象
  HitTestResult result = VisualTreeHelper.HitTest(canvasDrawingArea, pt);
  // 如果result不为null，说明鼠标确实单击了某个形状
  if (result != null)
  {
    // 获取被单击的那个形状，然后将其从画布中移除
    canvasDrawingArea.Children.Remove(result.VisualHit as Shape);
  }
}
```

该方法首先获取了用户单击Canvas的具体X、Y坐标，然后通过VisualTreeHelper.HitTest()方法执行了一个命中测试操作。如果用户没有单击Canvas中的某个UIElement，返回值HitTestResult对象将为null。如果HitTestResult不为null，我们可以通过VisualHit属性获取被单击的UIElement，然后将其转换为Shape派生类（记住，Canvas可以包含任何UIElement，而不仅仅是形状！）你将在下一章中看到有关“视觉树”的详细内容。

说明 默认情况下，VisualTreeHelper.HitTest()返回的是被单击的最顶层UIElement，而不会提供它下层的对象信息（如果对象按Z轴重叠）。

这样改动之后，你可以在画布中通过单击鼠标来添加一个形状，通过右击来删除一个形状。图29-3显示了这项功能：

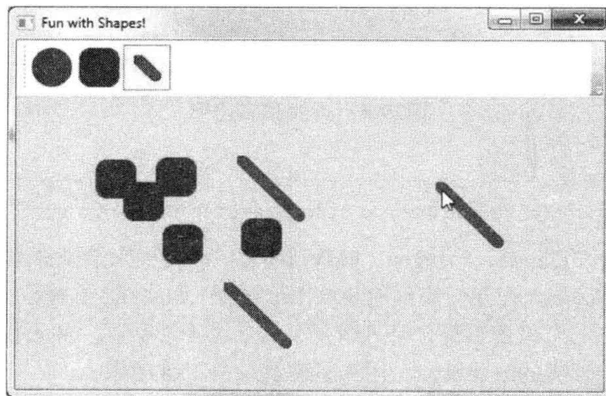


图29-3 形状之趣

在本节中，我们使用XAML在RadioButton中呈现了Shape派生类对象，并用C#填充了一个Canvas。在学习了画刷和图形变换之后，我们将给该示例添加更多的功能。为了演示UIElement对象的拖曳技术，我们还准备了一个新的示例。在这之前，我们先来学习一下System.Windows.Shapes的其他成员。

29.2.3　折线和多边形

上面的示例只用了3个Shape派生类。其他的子类（Polyine、Polygon和Path）如果没有工具（如Expression Blend或Expression Design）支持的话，要想正确呈现会显得极其烦琐。因为要表示这些形状的输出需要绘制大量的点。我们稍后将介绍Expression Design的作用，现在先来大致了解一下其他的Shape类型。

Polyline类型允许我们定义一个(x,y)坐标集合（通过Points属性），通过这些坐标绘制一系列线段，但它们不需要首尾相接。Polygon类型与之类似，只不过它会将起点和终点连接起来并用画刷进行填充。假设在kaxaml编辑器（或在第27章中创建的自定义XAML编辑器）中存在如下的<Stackpanel>:

```
<!-- Polylines不会自动闭合 -->
<Polyline Stroke ="Red" StrokeThickness ="20" StrokeLineJoin ="Round"
          Points ="10,10  40,40  10,90  300,50"/>
<!-- Polygon 总是自动闭合-->
<Polygon Fill ="AliceBlue" StrokeThickness ="5" Stroke ="Green"
         Points ="40,10 70,80 10,50" />
```

图29-4显示了输出结果。

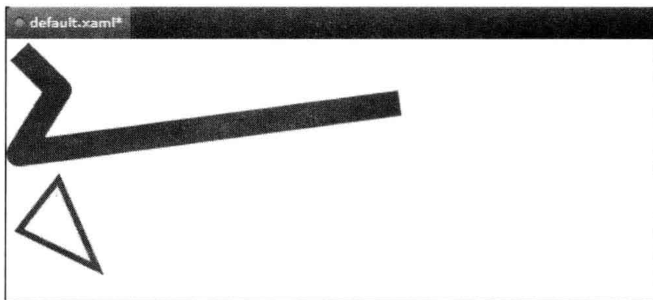


图29-4　折线和多边形

29.2.4　路径

单独使用Rectangle、Ellipse、Polygon、Polyline和Line类型来绘制详细的2D矢量图形可能会略显复杂，因为这些基元形状不允许我们简单地获取图形数据，如曲线、重叠数据，等等。最后一个Shape派生类Path通过相互独立的几何图形集合来表示复杂的二维图形数据。如果你定义了这样一个几何图形集合，可以将其赋给Path类的Data属性，用来呈现复杂的二维图像。

Data属性的类型为System.Windows.Media.Geometry派生类，其主要成员如表29-2所示。

表29-2 System.Windows.Media.Geometry类型的成员

成 员	含 义
Bounds	建立包含几何图形的边界矩形
FillContains()	判断给定的Point（或其他Geometry对象）是否包含在指定的Geometry派生类的边界内。这对于命中测试计算来说是十分有用的
GetArea()	返回Geometry派生类所占的整个区域
GetRenderBounds()	返回一个Rect，它所代表的矩形是用来呈现Geometry派生类的最小矩形
Tranform	设置一个用来改变几何图形呈现的Transform对象

Geometry的派生类（如表29-3所示）与Shape的派生类十分类似。例如，EllipseGeometry有着和Ellipse相似的成员。它们最大的区别就是Geometry派生类不知道如何直接呈现它们本身，并且它们不是继承自UIElement的。相反，Geometry派生类所表示的只不过是点数据的集合，这实际上是在说“如果将我赋给一个Path对象，我就会呈现我自己”。

说明 Path并不是WPF中唯一使用几何图形集合的类。例如，DoubleAnimationUsingPath、DrawingGroup、GeometryDrawing甚至UIElement都能分别使用各自的PathGeometry、ClipGeometry、Geometry和Clip属性来呈现几何图形。

表29-3 Geometry派生类

Geometry类	含 义
LineGeometry	表示一条直线
RectangleGeometry	表示一个矩形
EllipseGeometry	表示一个圆形
GeometryGroup	允许你将多个Geometry对象组合在一起
CombinedGeometry	允许你将两个不同的Geometry对象合并成一个单独的形状
PathGeometry	表示由直线和曲线组成的图像

假设kaxaml中有如下所示的Path，它使用了一些Geometry派生类。注意，我们将一个GeometryGroup对象赋给了Path的Data属性，这个GeometryGroup还包含了其他一些Geometry派生对象，如EllipseGeometry、RectangleGeometry和LineGeometry。输出结果如图29-5所示。

```
<!-- Path的Data属性值为一组几何图形对象 -->
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center = "75,70"
         RadiusX = "30" RadiusY = "30" />
      <RectangleGeometry Rect = "25,55 100 30" />
      <LineGeometry StartPoint="0,0" EndPoint="70,30" />
      <LineGeometry StartPoint="70,30" EndPoint="0,30" />
    </GeometryGroup>
  </Path.Data>
</Path>
```

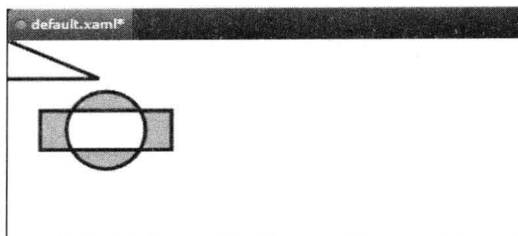


图29-5　包含不同Geometry对象的Path

图29-5中的图像完全可以用之前介绍的Line、Ellipse和Rectangle类来呈现。但这需要将各种UIElement对象放入内存。而如果使用几何图形来建立绘制点，然后将几何图形的集合放入可以呈现数据的容器中（如Path），则可以降低内存开销。

由于Path和其他System.Windows.Shapes有同样的继承链，可以像其他UIElement对象那样发送同样的事件通知。因此如果在Visual Studio项目中定义同样的<Path>元素，就可以编写鼠标的事件处理程序来判断用户是否单击了线段的某个区域（记住，kaxaml不允许在标记中处理事件）。

路径“迷你语言建模”

在表29-3所列的类中，不论是在XAML还是在代码中，PathGeometry都是最难配置的。PathGeometry中的每个段（segment）都是由包含其他段和图像的对象组成的（如ArcSegment、BezierSegment、LineSegment、PolyBezierSegment、PolyLineSegment、PolyQuadraticBezierSegment等）。下面的示例中包含一个Path对象，它的Data属性设置为一个由各种图像和段组成的<PathGeometry>。

```
<Path Stroke="Black" StrokeThickness="1" >
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,50">
          <PathFigure.Segments>
            <BezierSegment
              Point1="100,0"
              Point2="200,200"
              Point3="300,100"/>
            <LineSegment Point="400,100" />
            <ArcSegment
              Size="50,50" RotationAngle="45"
              IsLargeArc="True" SweepDirection="Clockwise"
              Point="200,100"/>
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

坦白地说，很少有程序员会通过直接描述Geometry或PathSegment派生类来手工创建复杂的二维图像。实际上，组成复杂的路径是通过Expression Design来完成的。

即使有这些工具的辅助，定义一个复杂Path对象的XAML代码还是十分庞大的，这是因为数据包含了各种Geometry或PathSegment派生类的完整描述。要创建更加简明紧凑的标记，可以选择为Path类设计的一种特定的“迷你语言”（mini-language）。

例如，我们不必用Geometry和PathSegment派生类的集合来设置Path的Data属性，而只是使用一个字符串。这个字符串包含一些特定的符号和各种值，可以定义要呈现的形状。事实上，用Expression工具创建Path对象时，会自动使用迷你语言。下面是一个简单的示例及其输出结果（如图29-6所示）。

```
<Path Stroke="Black" StrokeThickness="3"
     Data="M 10,75 C 70,15 250,270 300,175 H 240" />
```

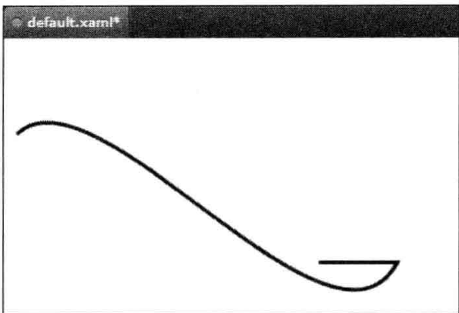


图29-6 路径迷你语言可以简洁地描述Geometry/PathSegment对象模型

M命令（move的简称）包含一个X、Y坐标来标识图形的起始点。C命令包含一系列绘制点来呈现曲线（确切地说是一个三次方贝赛尔曲线），而H绘制一个水平直线。

坦白地说，现在你手工创建或解析一个包含路径迷你语言指令的字符串的几率几乎为0。不过，至少你不会在看到由工具生成的XAML时会大吃一惊。如果你对这个特殊的语法感兴趣，可以查看.NET Framework 4.5 SDK文档中的“Path Markup Syntax”。

29.3 WPF 画刷和画笔

画刷可以控制2D表面的填充方式，它在各种WPF图形呈现选项（形状、绘图和几何图形、可视化）中被广泛地使用。WPF提供了6种不同的画刷类型，它们都扩展自System.Windows.Media.Brush。由于Brush是抽象的，填充一个区域主要使用表29-4所示的后代类。

29

表29-4 WPF画刷派生类型

画刷类型	含　义
DrawingBrush	使用Drawing派生对象（GeometryDrawing、ImageDrawing或VideoDrawing）绘制区域
ImageBrush	使用图像（ImageSource对象）绘制区域
LinearGradientBrush	使用线性渐变绘制区域
RadialGradientBrush	使用径向渐变绘制区域
SolidColorBrush	绘制纯色，设置其Color属性
VisualBrush	使用Visual派生对象（DrawingVisual、Viewport3DVisual和ContainerVisual）绘制区域

DrawingBrush和VisualBrush类允许你基于一个已知的Drawing或Visual派生类来构建画刷。这些画刷类可以用于其他两个WPF图形选项中（绘图和可视化），这将在本章稍后讨论。

顾名思义，ImageBrush允许你通过设置ImageSource属性来构建一个画刷，这个画刷能够显示外部文件或内嵌的应用程序资源中的图像数据。其余的画刷类型（LinearGradientBrush和RadialGradientBrush）尽管所需的XAML代码有点冗长，但用起来却十分简单。幸运的是，Visual Studio支持集成的画刷编辑器，大大简化了生成程式化画刷的过程。

29.3.1　使用Visual Studio配置画刷

现在我们使用一些更有趣的画刷来修改我们的WPF绘图程序RenderingWithShapes。我们之前绘制的3个形状使用了简单的纯色来呈现数据，因此可以通过简单的字符串文本来获取它们的值。不过为了让事情更有趣一些，我们将使用集成的画刷编辑器。确保IDE中打开的窗体为初始窗体的XAML编辑器，然后选择Ellipse元素。现在，在Properties窗体中找到Brush分类，点击最上面的Fill属性（如图29-7所示）。

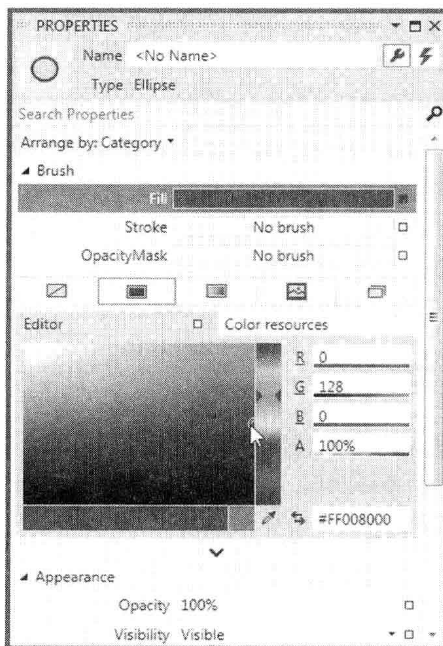


图29-7　任何使用画刷的属性都可以通过集成的画刷编辑器来进行配置

在画刷编辑器的最上方，是被选项中所有“画刷兼容的”属性（如Fill、Stroke、OpacityMash）。下面是一系列选项卡，可以配置不同的画刷类型，包括纯色画刷。你可以使用颜色选择器工具和ARGB（alpha、red、green和blue，其中alpha控制透明度）编辑器来控制当前画刷的颜色。使用这些滑块和相关的颜色选择区域，可以创建各种各样的纯色。我们继续使用这些工具来改变Ellipse的Fill颜色，并查看XAML的输出结果。你会发现颜色的值存储为十六进制，例如：

```
<Ellipse Fill="#FF47CE47" Height="35" Width="35" />
```

更有趣的是，同样的编辑器还允许你配置渐变色的画刷，可以定义一系列颜色和变换点。画刷编

辑器包含一组选项卡，其中第一个将设置为空刷（null brush），不会呈现任何内容。其余4个可分别用来设置纯色画刷（我们刚刚讨论过）、渐变色画刷、瓦片画刷和图像画刷。

单击渐变色画刷按钮，编辑器将显示一些新的选项（如图29-8所示）。左下方的 3 个按钮可以选择线性渐变、径向渐变或反转渐变截止点。最下端的滑块显示各个渐变截止点的颜色，每个截止点由滑块上的一个“箭头”标记。你可以通过拖动滑块上的箭头来控制渐变的偏移量。此外，单击某个箭头，还可以在颜色选择器上来改变渐变截止点的颜色。最后，直接单击渐变滑块，可以添加一个新的渐变截止点。

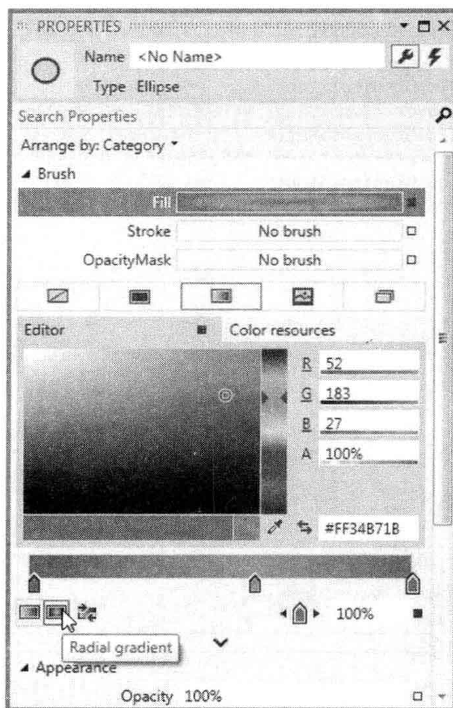


图29-8　使用Visual Studio画刷编辑器创建基本的渐变画刷

现在我们来花点时间在这个编辑器上创建一个径向渐变画刷，它包含三个渐变截止点，分别设置为你所选择的颜色。图29-8显示了我刚刚构建的画刷，使用了三种深浅不同的绿色。

完成之后，IDE将更新XAML，生成一个自定义的画刷并通过属性元素语法将其赋给一个与画刷兼容的属性（本例中Ellipse的Fill属性）。例如：

```
<Ellipse Height="35" Width="35">
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="#FF87E71B" Offset="0.589" />
      <GradientStop Color="#FF2BA92B" Offset="0.013" />
      <GradientStop Color="#FF34B71B" Offset="1" />
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

29.3.2　在代码中配置画刷

由于我们为Ellipse的XAML构建了一个自定义画刷，相应的C#代码就过期了，因为它所呈现的仍然是纯绿色的圆圈。要想在代码中达到同样的效果，需要用刚刚创建的画刷来更新代码。下面的代码显示了必要的更新，它看上去比想象的要复杂一些，这是因为我们使用了System.Windows.Media.ColorConverter类将十六进制的值转换成了Color对象（图29-9为更改后的输出结果）：

```
case SelectedShape.Circle:
  shapeToRender = new Ellipse() { Height = 35, Width = 35 };

  // 在代码中创建一个RadialGradientBrush
  RadialGradientBrush brush = new RadialGradientBrush();
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF87E71B"), 0.589));
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF2BA92B"), 0.013));
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF34B71B"), 1));

  shapeToRender.Fill = brush;
  break;
```

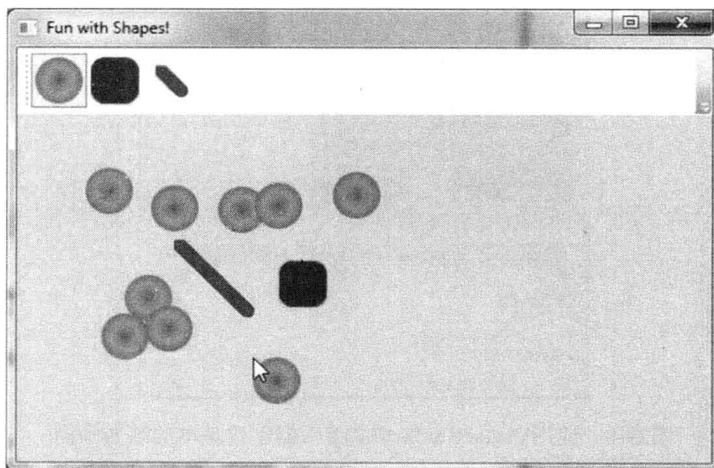


图29-9　用更漂亮的方式来绘制圆圈

顺便说一句，如果在创建GradientStop时希望使用简单的颜色，可以将构造函数的第一个参数设置为Colors枚举，它能返回配置好的Color对象：

```
GradientStop g = new GradientStop(Colors.Aquamarine, 1);
```

或者，如果你需要对颜色进行细微地控制，可以传入一个特定的Color对象。例如：

```
Color myColor = new Color() { R = 200, G = 100, B = 20, A = 40 };
GradientStop g = new GradientStop(myColor, 34);
```

当然，Colors枚举和Color类并不仅限于渐变画刷。在代码中你可以将其用于任何需要表示一个颜色的地方。

29.3.3 配置画笔

与画刷相比，画笔是用来绘制几何图形边框或线条图形（如Line、PolyLine类）的对象。Pen类允许我们绘制指定粗度，用double值表示。此外，Pen的一些属性还与Shape类的属性类似，如开始和结束的笔端样式、虚线模式，等等。例如：

```
<Pen Thickness="10" LineJoin="Round" EndLineCap="Triangle" StartLineCap="Round" />
```

在大多数情况下，你都不需要直接创建Pen对象，因为指定某个属性的值将会间接创建，如Shape派生类（以及其他UIElement）的StrokeThickness属性。尽管如此，在使用Drawing派生类型（本章稍后会有介绍）时，创建一个自定义的Pen对象还是十分方便的。Visual Studio没有画笔编辑器，但它允许你在Properties窗体中配置被选项中所有与笔画相关的属性。

29.4 图形变换

让我们通过变换这个话题来对形状的讨论做一个总结。WPF发布了很多扩展自System.Windows.Media.Transform抽象基类的类。表29-5列出了很多主要的Transform派生类。

表29-5 System.Windows.Media.Transform类型的主要后代

类　型	含　义
MatrixTransform	创建任意一个矩阵变换，用于操作2D平面中的对象或坐标系
RotateTransform	在2D(*x*,*y*)坐标系内围绕一个指定点，按顺时针方向旋转对象
ScaleTransform	在2D(*x*,*y*)坐标系内缩放一个对象
SkewTransform	在2D(*x*,*y*)坐标系内扭曲一个对象
TranslateTransform	在2D(*x*,*y*)坐标系内平移（移动）一个对象
TransformGroup	标识由其他Transform对象组合成的复合Transform

你可以对任何UIElement应用变换（如Shape派生类以及Button、TextBox等控件）。使用这些变换类，你可以用给定的角度呈现图形数据，可以扭曲平面中的图像，还可以扩大、缩小或反转目标项。

说明 尽管变换对象可以用于各个地方，但你会发现最常用的还是WPF动画和自定义控件模板。正如你稍后会看到的那样，可以使用WPF动画为自定义控件添加一些视觉效果。

变换对象可以赋给包含两个共同属性的目标对象（如Button、Path等）。LayoutTransform属性用于将元素呈现到布局管理器之前发生变换，因此它不会对Z轴的操作产生影响（也就是说，发生变换的图像数据不会产生重叠）。

RenderTransform属性发生于元素呈现在容器中之后，因此，根据元素在容器中的位置，它们在进行变换时很有可能相互重叠。

29.4.1　变换概览

稍后我们将在RenderingWithShapes项目中加入一些变换逻辑。现在，我们先来看看实际的变换对象。打开kaxaml（或自定义的XAML编辑器），在根元素<Page>或<Window>中定义一个简单的<StackPanel>，将其Orientation属性设置为Horizontal。然后，添加下面的<Rectangle>，它使用RotateTransform对象旋转了45°。

```
<!-- 进行旋转变换的矩形 -->
<Rectangle Height ="100" Width ="40" Fill ="Red">
  <Rectangle.LayoutTransform>
    <RotateTransform Angle ="45"/>
  </Rectangle.LayoutTransform>
</Rectangle>
```

下面是一个<Button>，它使用<SkewTransform>对表面扭曲了20%。

```
<!-- 进行扭曲变换的按钮 -->
<Button Content ="Click Me!" Width="95" Height="40">
  <Button.LayoutTransform>
    <SkewTransform AngleX ="20" AngleY ="20"/>
  </Button.LayoutTransform>
</Button>
```

此外，还有一个通过ScaleTransform放大20倍的<Ellipse>（注意其初始的Height和Width值），以及使用了多个变换对象的<TextBox>。

```
<!-- 放大了20倍的圆形-->
<Ellipse Fill ="Blue" Width="5" Height="5">
  <Ellipse.LayoutTransform>
    <ScaleTransform ScaleX ="20" ScaleY ="20"/>
  </Ellipse.LayoutTransform>
</Ellipse>

<!-- 旋转和扭曲了的文本框 -->
<TextBox Text ="Me Too!" Width="50" Height="40">
  <TextBox.LayoutTransform>
    <TransformGroup>
      <RotateTransform Angle ="45"/>
      <SkewTransform AngleX ="5" AngleY ="20"/>
    </TransformGroup>
  </TextBox.LayoutTransform>
</TextBox>
```

要注意的是在应用了某个变换后，你不需要执行任何手工的计算来正确响应命中测试、输入焦点等操作。WPF图形引擎已经为你处理好了这一切。例如在图29-10中，你可以看到TextBox仍然可以响应键盘输入。

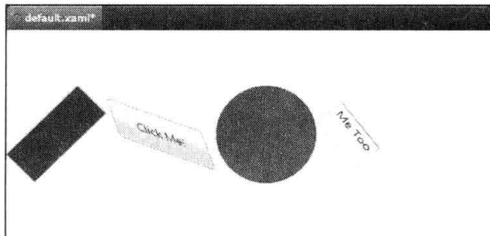


图29-10　图形变换对象的输出结果

29.4.2 变换Canvas数据

现在我们来看看如何在RenderingWithShapes示例中加入一些变换逻辑。除了可以对一个单个项（如Rectangle、TextBox等）进行变换之外，你还可以对一个布局管理器使用变换对象，这可以对其内部的所有数据进行变换。例如，你可以对主窗体的整个<DockPanel>旋转一定的角度：

```
<DockPanel LastChildFill="True">
  <DockPanel.LayoutTransform>
    <RotateTransform Angle="45"/>
  </DockPanel.LayoutTransform>
...
</DockPanel>
```

这个例子有点极端，现在我们来添加最后一个（不那么极端的）特性，允许用户反转整个Canvas及其包含的图形。我们向<ToolBar>中添加一个<ToggleButton>，如下所示：

```
<ToggleButton Name="flipCanvas" Click="flipCanvas_Click" Content="Flip Canvas!"/>
```

在Click事件处理程序中，创建一个RotateTransform对象，在单击这个新的ToggleButton时，通过LayoutTransform属性将其与Canvas对象进行关联。要移除变换可以将该属性设置为null：

```
private void flipCanvas_Click(object sender, RoutedEventArgs e)
{
  if (flipCanvas.IsChecked == true)
  {
    RotateTransform rotate = new RotateTransform(-180);
    canvasDrawingArea.LayoutTransform = rotate;
  }
  else
  {
    canvasDrawingArea.LayoutTransform = null;
  }
}
```

运行程序，在画布区域中添加一些图形。单击新的按钮，你会发现这些形状数据流出了画布边界！这是因为我们还没有定义剪裁区域（如图29-11所示）。

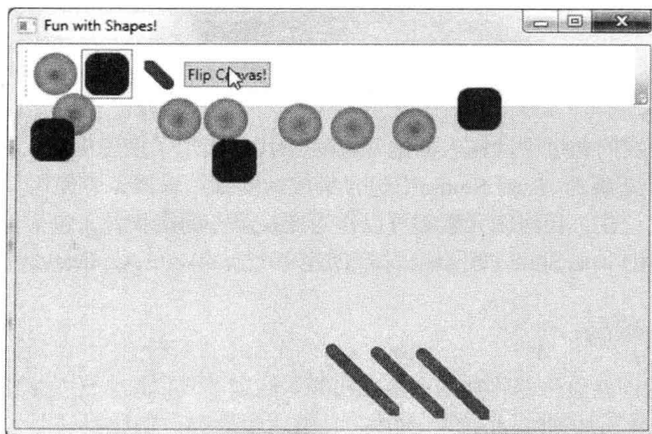


图29-11 啊！数据在变换后流到了画布外面

要修复这个问题是很容易的。你不必手工编写复杂的剪裁逻辑，只需要将<Canvas>的ClipToBounds属性设置为true，这会避免子元素呈现到父元素边界之外。再次运行程序，你会发现数据不会再跑到画布外面了。

```
<Canvas ClipToBounds = "True" ... >
```

还有一个小问题需要修改。当你按下切换按钮翻转画布后，再次单击画布绘制新的形状时，你单击的点并不是图形数据实际绘制的点。数据已经呈现在鼠标光标处了。

要解决这个问题，我在代码中添加了一个Boolean成员变量（isFlipped），它将在呈现发生之前对绘制的形状执行同样的变换（通过RenderTransform）。代码如下：

```
private void canvasDrawingArea_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
  Shape shapeToRender = null;
...

  // isFlipped为私有的布尔字段，单击切换按钮将改变它的状态
  if (isFlipped)
  {
    RotateTransform rotate = new RotateTransform(-180);
    shapeToRender.RenderTransform = rotate;
  }

  // 设置画布的top/left
  Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
  Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);
  // 绘制形状
  canvasDrawingArea.Children.Add(shapeToRender);
}
```

这个示例包含了System.Windows.Shapes、画刷和变换。在学习使用绘图和几何图形呈现数据之前，我们先来看看如何使用Expression Blend来简化基元图形的使用。

源代码　RenderingWithShapes项目的源代码位于Chapter 29子目录下。

29.5　使用 Visual Studio 变换编辑器

在前面的示例中，我们通过手动输入标记并编写C#代码实现了多种变换。尽管这很有用，但你一定会非常高兴地看到最新版的Visual Studio中包含集成的变换编辑器，它使用集成的工具非常轻松地生成必要的转换标记。记住，任何UI元素都可以作为变换服务的接收者，包括包含不同UI元素的布局系统。为了演示如何使用Visual Studio变换编辑器，创建一个新的WPF Application项目FunWithTrasforms。

29.5.1　构建初始布局

首先，使用集成的网格编辑器将初始的Grid切分为两列（具体尺寸无关紧要）。现在，在Toolbox中找到StackPanel控件并添加到Grid的第一列中，使其占满该列。例如：

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="221*"/>
    <ColumnDefinition Width="288*"/>
  </Grid.ColumnDefinitions>
  <StackPanel HorizontalAlignment="Left" Height="292" Margin="10,10,0,0"
              VerticalAlignment="Top" Width="201"/>
</Grid>
```

下面，在Document Outline面板中选择StackPanel，向StackPanel容器中添加3个Button控件（如图29-12所示）。

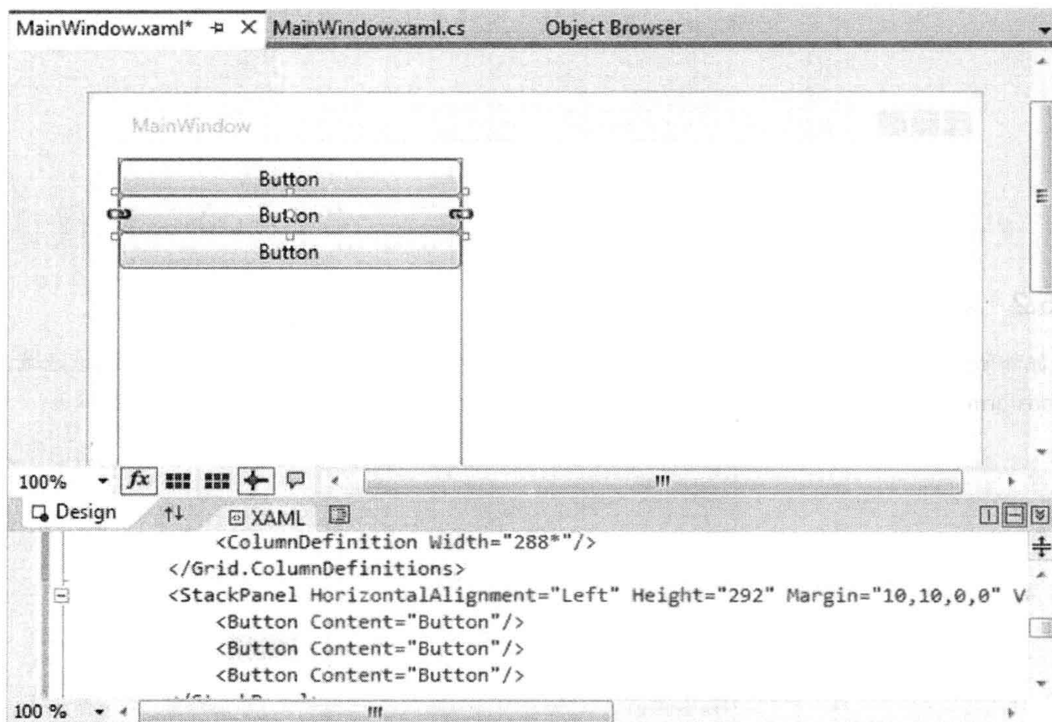


图29-12　包含Button控件的StackPanel

分别选中3个Button，将其Content属性（位于Properties窗口的Common 部分）分别改为Skew、Rotate和Flip。同样使用Properties面板中的Name域为每个按钮起一个合适的名字，如btnSkew、btnRotate和btnFile。使用Properties面板中的Events选项卡，处理每个Button的Click事件。我们稍后来实现这些处理程序。

最后，在Grid第二列中创建一个你选择的图像（可以使用本章介绍的任何技术）。图29-13显示了最终的布局。我在这里创建了两个Ellipse控件，并将其组合进一个名为myCanvas的Canvas控件。

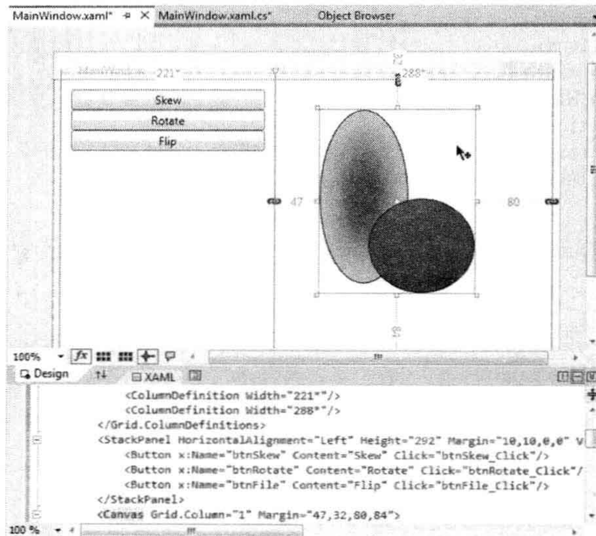


图29-13　变换示例的布局

29.5.2　在设计时应用变换

如前所述，Visual Studio提供了一个集成的变换编辑器，位于Properties面板中。找到该区域，展开Transform节可以看到编辑器中的RenderTransform和LayoutTransform子节（如图29-14所示）。

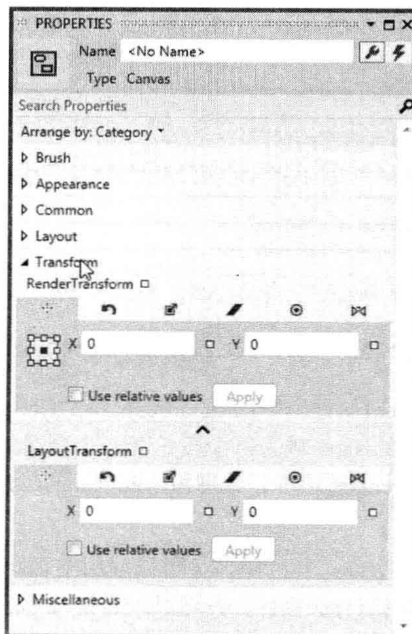


图29-14　变换编辑器

和Brushes节类似，Transform节提供了很多选项卡，可以为当前选中项配置不同的图像变化类型。表29-6描述了各个变换选项，按选项卡从左到右的顺序列出。

表29-6 变换选项

类 型	含 义
Translate（平移）	通过设置X, Y来平移某项的位置
Rotate（旋转）	通过设置角度来旋转某项
Scale（缩放）	通过设置X, Y来缩放某项
Skew（扭曲）	通过设置X, Y方向来扭曲选中项的边框
Center Point（中心点）	当旋转或翻转对象时，对象的移动是围绕一个相对固定的点进行的，叫做对象的中心点。默认情况下，对象的中心点位于对象的中心，但这种变换可以改变对象的中心点，使旋转或翻转时围绕的是一个不同的点
Flip（翻转）	基于X或Y中心点来翻转选中的项

建议你用自定义形状作为目标，逐个测试这些变化（用Ctrl+Z取消之前的操作）。和Transform Properties面板的其他部分一样，每个变换节点都有不同的配置选项，它们都是非常容易理解的。例如，Skew变换编辑器可以设置X和Y扭曲值，Flip变换编辑器可以按X轴或Y轴翻转，等等。

29.5.3 在代码中变换画布

每个Click事件处理程序或多或少都十分类似。我们会配置一个变换对象，并将其赋值给myCanvas对象。因此运行应用程序时，点击按钮就可以看到变换的结果。以下是各个事件处理程序的完整代码（注意我设置的是LayoutTransform属性，因此形状数据的位置是相对于父容器来说的）：

```
private void btnFlip_Click(object sender, System.Windows.RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new ScaleTransform(-1, 1);
}
private void btnRotate_Click(object sender, System.Windows.RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new RotateTransform(180);
}
private void btnSkew_Click(object sender, System.Windows.RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new SkewTransform(40, -20);
}
```

源代码 FunWithTransformations项目的源代码位于Chapter29子目录下。

29.6 使用绘图和几何图形呈现图形数据

尽管Shape类型可以用来生成各种二维平面图形，但其丰富的继承链需要消耗大量的内存。尽管Path类可以通过内含几何图形的方式（而非大量的其他形状）来减少一部分内存损耗，不过WPF还是

提供了一个高雅的绘图和几何图形编程接口，可以呈现更轻量级的2D矢量图像。

该API的入口点是System.Windows.Media.Drawing抽象类（位于PresentationCore.dll中），它本身也是通过定义一个矩形边界来呈现的。注意在图29-15中，Drawing类的继承链比Shape要精简得多，UIElement和FrameworkElement都不在其中。

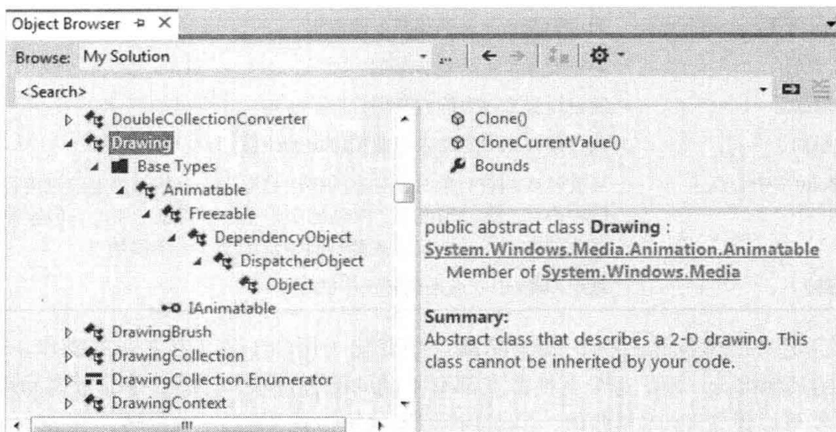


图29-15　Drawing类比Shape要精简得多

WPF提供了很多Drawing的扩展类。如表29-7所示，它们都代表了一种特殊的绘制内容的方式。

表29-7　WPF中的Drawing派生类型

类　型	含　义
DrawingGroup	将多个独立的Drawing派生对象合并成一个单独的复合绘图进行呈现
GeometryDrawing	用轻量级的方式呈现2D形状
GlyphRunDrawing	用WPF图形呈现服务呈现文本数据
ImageDrawing	在矩形边界内呈现图像文件或几何图形集
VideoDrawing	播放音频或视频文件。该类型只能使用C#代码访问。如果要在XAML中播放视频，最好使用MediaPlayer类型

由于Drawing是轻量级的，没有继承UIElement或FrameworkElement，因此其派生类型不能处理输入事件（尽管它们可以以编程方式执行命中测试逻辑）。

Drawing派生类型与Shape派生类型另一个关键的不同点是，由于Drawing没有继承UIElement，因此不能进行呈现。要想显示它们的内容，只能将它们放置到一个宿主对象中（如DrawingImage、DrawingBrush或DrawingVisual）。

我们可以使用DrawingImage将绘图和几何图形放置到WPF的Image控件中，该控件通常用来显示外部文件中的数据。DrawingBrush可以用来构建一个基于绘图和几何图形的画刷，来设置需要画刷的属性。最后，DrawingVisual仅用于图形呈现的完全由C#代码驱动的“可视化”层。

尽管绘图比形状要稍显复杂，但它将图形的组成和图形的呈现解耦，这使得Drawing派生类型在保留关键服务的同时比Shape派生类型要更加轻便。

29.6.1 使用几何图形构建DrawingBrush

在前面的示例中，我们用多个几何图形填充一个Path，例如：

```
<Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center = "75,70"
         RadiusX = "30" RadiusY = "30" />
      <RectangleGeometry Rect = "25,55 100 30" />
      <LineGeometry StartPoint="0,0" EndPoint="70,30" />
      <LineGeometry StartPoint="70,30" EndPoint="0,30" />
    </GeometryGroup>
  </Path.Data>
</Path>
```

这样，我们通过Path获得了交互功能，并且由于使用了几何图形，因此仍然是非常轻便的。但是，如果你希望呈现相同的内容却不需要任何交互性，可以将相同的<GeometryGroup>放置到一个DrawingBrush内，例如：

```
<DrawingBrush>
  <DrawingBrush.Drawing>
    <GeometryDrawing>
      <GeometryDrawing.Geometry>
        <GeometryGroup>
          <EllipseGeometry Center = "75,70"
                           RadiusX = "30" RadiusY = "30" />
          <RectangleGeometry Rect = "25,55 100 30" />
          <LineGeometry StartPoint="0,0" EndPoint="70,30" />
          <LineGeometry StartPoint="70,30" EndPoint="0,30" />
        </GeometryGroup>
      </GeometryDrawing.Geometry>
      <!-- 用来绘制边框的自定义画笔 -->
      <GeometryDrawing.Pen>
        <Pen Brush="Blue" Thickness="3"/>
      </GeometryDrawing.Pen>
      <!-- 用来填充内部的自定义画刷 -->
      <GeometryDrawing.Brush>
       <SolidColorBrush Color="Orange"/>
      </GeometryDrawing.Brush>
    </GeometryDrawing>
  </DrawingBrush.Drawing>
</DrawingBrush>
```

当你将多个几何图形放置到DrawingBrush内时，你还需要建立一个Pen对象来绘制边框，因为我们没有从Shape基类那里继承Stroke属性。这里，我创建了一个<Pen>，其设置与前面Path示例中的Stroke和StrokeThickness的值一样。

此外，由于没有从Shape继承Fill属性，我们还需要用属性元素语法来为<GeometryDrawing>定义一个画刷对象。这里的设置和前面的Path设置相同，使用了一个橙色的画刷。

29.6.2 用DrawingBrush进行绘画

有了DrawingBrush，就可以将它赋值给任何需要画刷对象的属性了。例如，如果在kaxaml中存在如下的代码，就可以使用属性元素语法在Page的整个表面进行绘图：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Page.Background>
    <!-- 和上例一样的DrawingBrush  -->
    < DrawingBrush >
      ...
    </DrawingBrush >
  </Page.Background>
</Page>
```

或者，你也可以将该<DrawingBrush>赋值给画刷兼容的属性，如Button的Background属性：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Button Height="100" Width="100">
    <Button.Background>
      <!--和上例一样的DrawingBrush -->
      </ DrawingBrush >
        ...
      </ DrawingBrush >
    </Button.Background>
  </Button>

</Page>
```

重要的是，无论将该自定义的<DrawingBrush>赋值给哪个画刷兼容的属性，这种呈现2D矢量图像的损耗要比使用形状呈现同样的2D图像小得多。

29.6.3 在DrawingImage中使用绘图类型

使用DrawingImage类型可以将几何图形放置于WPF的<Image>控件中。例如：

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Image Height="100" Width="100">
    <Image.Source>
      <DrawingImage>
        <DrawingImage.Drawing>
          <GeometryDrawing>
            <GeometryDrawing.Geometry>
              <GeometryGroup>
                <EllipseGeometry Center = "75,70"
                                 RadiusX = "30" RadiusY = "30" />
                <RectangleGeometry Rect = "25,55 100 30" />
                <LineGeometry StartPoint="0,0" EndPoint="70,30" />
                <LineGeometry StartPoint="70,30" EndPoint="0,30" />
              </GeometryGroup>
            </GeometryDrawing.Geometry>

            <!-- 用来绘制边框的自定义画笔 -->
            <GeometryDrawing.Pen>
              <Pen Brush="Blue" Thickness="3"/>
            </GeometryDrawing.Pen>
```

```
            <!-- 用来填充内部的自定义画刷 -->
            <GeometryDrawing.Brush>
              <SolidColorBrush Color="Orange"/>
             </GeometryDrawing.Brush>
           </GeometryDrawing>
         </DrawingImage.Drawing>
       </DrawingImage>
     </Image.Source>
   </Image>
 </Page>
```

在本例中，我们的<GeometryDrawing>没有放置在<DrawingBrush>里，而是放置在一个<DrawingImage>中。我们可以使用<DrawingImage>来设置Image控件的Source属性。

29.7 Expression Design 的作用

毫无疑问，一个美术设计师用Visual Studio提供的工具和技术创建复杂的矢量图形是非常具有挑战性的。幸运的是，我们可以使用微软Expression Design创建异常复杂的图像数据，并导出为多种文件格式，包括XAML。如果数据导出为XAML，就可以很容易地导出为WPF。这样就可以使用Visual Studio对生成的对象模型进行编程。

说明 下面的示例应用程序将演示如何将图像数据导出为XAML，并将这些标记导入到新的Visual Studio WPF应用程序中。但如果你没有Expression Blend也不要担心。在下载的代码文件的Chapter29文件夹下有必要的bear_paper.xaml文件。

29.7.1 将示例设计文件导出为XAML

在向新的WPF应用程序导入复杂图像数据之前，首要任务是生成所谓的图像数据。启动Expression Design。如果你具备绘图技能，可以为本例绘制自定义图像。但如果你像我一样是个美术设计盲，可以简单地激活Help→Samples菜单选项。你将发现许多不同的可加载的*.design文件，如bear_paper.design（如图29-16所示）。

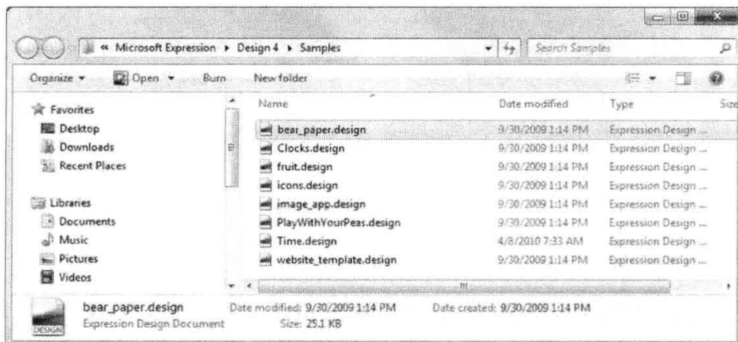


图29-16　泰迪熊示例图像

说明　Expression Design的介绍超出了本书范畴（坦白地说，也超出了我目前的能力）。如果有兴趣学习更多使用Design创建自定义图像的知识，可是使用Help菜单启动Expression Design User Guide。

这时，你会看到一张泰迪熊的图片出现在Expression Design的设计面板上。按下Ctrl+A组合键选中整个图片，改变图片数据的尺寸让其占满整个视口（viewport），如图29-17所示。

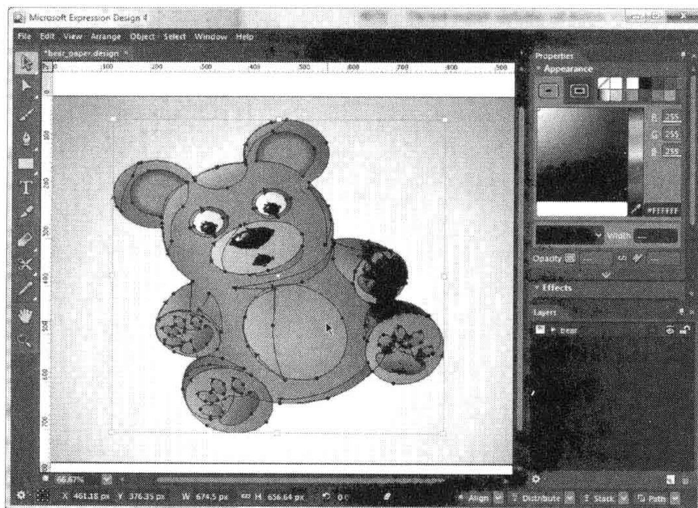


图29-17　改变泰迪熊图像的尺寸

在图像数据最终完成后，我们可以使用File→ Export...菜单选项将其导出全部或者部分数据。在Format列表框中可以看到很多流行的文件格式，但对于本例来说，我们选择XAML Silverlight 4/WPF Canvas选项（如图29-18所示）。

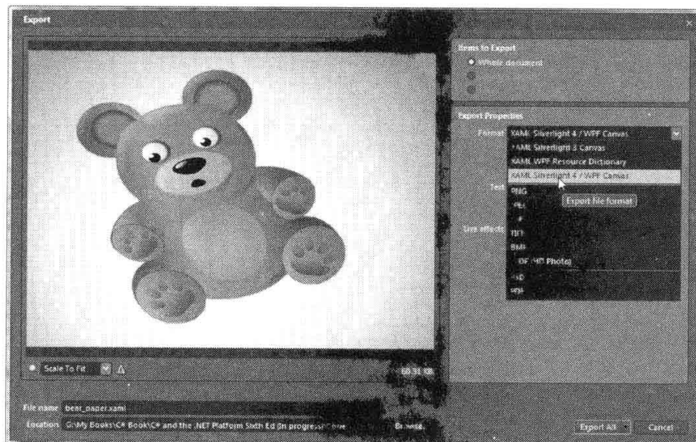


图29-18　将泰迪熊示例图像导出为XAML

选择了正确的XAML文件格式之后，取消Always name object选项。要记住当元素包含x:Name特性的时候，就可以在代码中与该项交互。但这个泰迪熊将使用大量XAML元素来描述。如果让工具为每个可能的对象命名，就会导致在C#代码库中添加大量的用不到的成员变量，（这也会增加最终编译的应用程序的尺寸）。

还要为这个导出的数据（bear_paper.xaml）选择一个轻易就能找到的位置（如Windows桌面）。其他选项都保留默认值。准备就绪后，点击Export All按钮，看，泰迪熊图片数据已经导出为XAML了。这时可以关闭Expression Design了。

29.7.2 将图像数据导入WPF对象

这时，我们可以使用Visual Studio构建一个新的WPF应用程序（取名为InteractiveTeddyBear）。完成之后，将初始窗体的大小改为如下所示的Height和Width，并删除初始的Grid控件：

```
<Window x:Class="InteractiveTeddyBear.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="824" Width="1056">
</Window>
```

现在，激活Project→ Add Existing Item菜单选项，使用弹出的对话框，导航到导出的bear_paper.xaml文件所在的位置。点击Add按钮，会发现该XAML文件已经添加到项目中了（可以检查Solution Explorer面板验证这一点）。双击该文件，将泰迪熊图像数据加载到Visual Studio设计器上。

查看bear_paper.xaml文件的Document Outline编辑器，会发现所有XAML元素无一缺席。我们的目标是找到熊的左眼球和右内耳，并为它们命名。你可以手动找到正确的对象（这个过程可能很无聊），使用可视设计器简单地点击这些项就能找到它们。Document Outline编辑器中相应的节点将自动高亮。选择左眼球，使用Properties编辑器将该对象命名为leftEye。选中右耳的一部分（随便哪一部分都行），命名为rightEar。图29-19展示了这个基本过程。

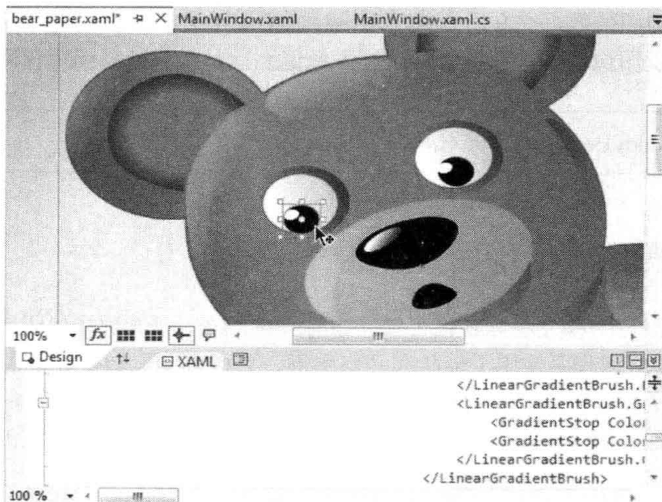


图29-19 找出对象并命名

现在，将整个XAML Canvas复制到剪切板上。这时可以关闭bear_paper.xaml文件。将设计器切换回初始窗体，将Canvas数据粘贴到开放和闭合的<Window>元素之间。现在可以看到泰迪熊数据显示在应用程序的主窗体中。如果愿意，可以随意调整图像在布局管理器中的确切位置，只要左眼球和右耳朵可见，就可以随时在代码中与这些对象交互。

29.7.3　与熊共舞

这时你就可以为leftEye和rightEar对象处理事件了。在设计器中选择对象，激活Properties窗口的Events区域，按要求输入事件处理程序的名称。在本例中，我们处理这两个对象的MouseLeftButtonDown事件，每次指定一个方法名。

下面这段简单的代码将在点击鼠标时改变每个对象的外观（如果你不愿意输入下面的所有代码，可以简单地在每个处理程序中添加一条MessageBox.Show()语句，并显示适当的信息）。

```
private void leftEye_MouseLeftButtonDown(object sender,
  System.Windows.Input.MouseButtonEventArgs e)
{
  //在点击时改变眼睛的颜色
  leftEye.Fill = new SolidColorBrush(Colors.Red);
}
private void rightEar_MouseLeftButtonDown(object sender,
  System.Windows.Input.MouseButtonEventArgs e)
{
  //在点击时使耳朵变得模糊
  System.Windows.Media.Effects.BlurEffect blur =
    new System.Windows.Media.Effects.BlurEffect();
  blur.Radius = 10;
  rightEar.Effect = blur;
}
```

现在运行应用程序。点击熊的左眼和右耳查看效果。

现在你已经理解了将Expression Design创建的复杂图像数据导入到Visual Studio的过程了，更重要的是，如何在代码中与图像数据交互。专业美术设计人士生成复杂图像数据并导出为XAML的能力是多么强大。一旦生成了图像数据，开发者可以导入标记并针对对象模型编写程序。

源代码　InteractiveTeddyBear项目的源代码位于Chapter 29子目录下。

29.8　使用可视化层呈现图形数据

WPF呈现图形的最后一种方法为可视化层。如前所述，该层只能通过代码访问（它不是XAML友好的）。绝大多数WPF应用程序使用了形状和几何图形，它们都能运转良好。而可视化层则提供了最快的方式来呈现大量的图形数据。奇妙的是，这个非常底层的图形层在向较大的区域呈现单个图像时也是非常有用的。例如，如果你需要用一个静态图像来填充窗体的背景，可视化层提供了最快速的方式，或者如果你希望基于用户输入之类的方式来快速切换窗体背景，可视化层也是十分有用的。

我们不会花太多时间来研究可视化层的详细内容，我们只建立一个很小的示例来演示基本信息。

1. Visual基类和派生类

抽象的System.Windows.Media.Visual类型只提供了用于呈现图像的部分服务（如呈现、命中测试、变换），而没有提供其他可以让代码膨胀的非可视化服务（如输入事件、布局服务、样式和数据绑定）。注意如图29-20所示的Visual类型的简单继承链。

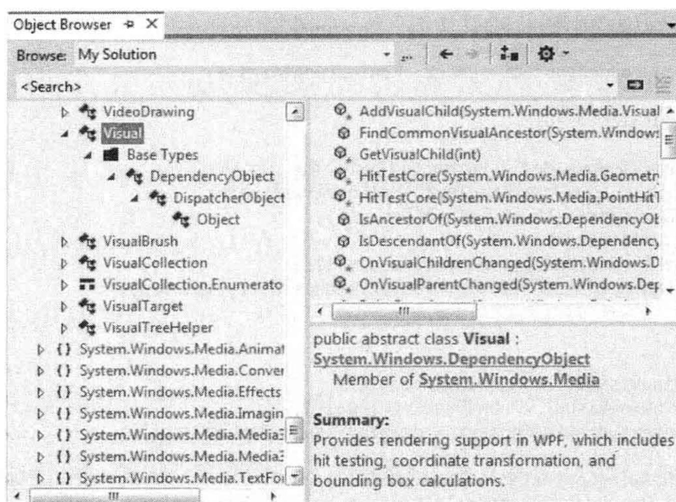


图29-20　Visual类型提供了基本的命中测试、坐标变换和边界框计算

由于Visual是抽象基类，因此我们需要使用它的派生类来执行实际的呈现操作。WPF提供了几个子类，如DrawingVisual、Viewport3DVisual和ContainerVisual。

在本例中，我们只关注DrawingVisual，它是一个轻量级的绘图类，可以用来呈现形状、图像或文本。

2. DrawingVisual类概览

要使用DrawingVisual在平面上呈现数据，需要执行以下基本步骤：

- 从DrawingVisual中获取DrawingContext对象；
- 使用DrawingContext呈现图形数据。

这是将数据呈现到平面所需要的最基本的两个步骤。但是，如果你希望被呈现的图形数据能够响应命中测试计算（这对添加用户交互功能来说是非常重要的），你需要执行以下额外的步骤：

- 更新你所呈现的容器中的逻辑树和可视化树；
- 重写FrameworkElement类中的两个虚方法，允许容器获取你创建的可视化数据。

我们来稍微讨论一下最后这两个步骤。首先，为了演示如何使用DrawingVisual类呈现2D数据，使用Visual Studio新建一个WPF应用程序RenderingWithVisuals。我们的首要目标是使用DrawingVisual为WPF Image控件动态设置数据。将窗体的XAML修改为如下形式：

```
<Window x:Class="RenderingWithVisuals.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title=" Fun with the Visual Layer" Height="350" Width="525"
```

29

```
        Loaded="Window_Loaded" WindowStartupLocation="CenterScreen">
    <StackPanel Background="AliceBlue" Name="myStackPanel">
        <Image Name="myImage" Height="80"/>
    </StackPanel>
</Window>
```

注意<Image>控件没有Source值，我们将在运行时进行设置。我们还指定了窗体的Loaded事件，将使用DrawingBrush对象在内存中构建图形数据。Loaded事件处理程序的实现如下所示：

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  const int TextFontSize = 30;

  // 创建一个System.Windows.Media.FormattedText对象
  FormattedText text = new FormattedText("Hello Visual Layer!",
    new System.Globalization.CultureInfo("en-us"),
    FlowDirection.LeftToRight,
    new Typeface(this.FontFamily, FontStyles.Italic,
                 FontWeights.DemiBold, FontStretches.UltraExpanded),
    TextFontSize,
    Brushes.Green);

  // 创建一个DrawingVisual，并获取DrawingContext
  DrawingVisual drawingVisual = new DrawingVisual();
  using(DrawingContext drawingContext = drawingVisual.RenderOpen())
  {
    // 现在，调用DrawingContext中的方法来呈现数据
    drawingContext.DrawRoundedRectangle(Brushes.Yellow, new Pen(Brushes.Black, 5),
      new Rect(5, 5, 450, 100), 20, 20);
    drawingContext.DrawText(text, new Point(20, 20));
  }

  // 使用DrawingVisual中的数据动态创建位图
  RenderTargetBitmap bmp = new RenderTargetBitmap(500, 100, 100, 90,
    PixelFormats.Pbgra32);
  bmp.Render(drawingVisual);

  // 设置Image控件的源
  myImage.Source = bmp;
}
```

这段代码引入了一些新的WPF类，我将进行简要的介绍（如果感兴趣，可以查看.NET Framework 4.5 SDK文档）。方法的开始部分创建了一个FormattedText对象，用来表示我们构建的内存图像的文本部分。如你所见，该构造函数可以指定很多特性，如字体大小、字体族群、前景颜色和文本本身。

接下来，我们调用DrawingVisual实例的RenderOpen()方法来获取必需的DrawingContext对象。在这里，我们在DrawingVisual中呈现了一个纯色的圆角矩形，以及之前创建的格式化文本。我们通过硬编码的方式将这两个图形数据添加到DrawingVisual中，这在生产中不见得是个好主意，但对于我们的测试来说是无关紧要的。

说明　在.NET Framework 4.5 SDK文档中查看DrawingContext类的所有呈现成员。如果你过去使用过Windows Forms图形对象，DrawingContext看起来会非常眼熟。

最后的几条语句将DrawingVisual映射到RenderTragetBitmap对象，后者位于System. Window.Media. Imaging命名空间，可将一个可视化对象转换为内存中的位图。然后，我们设置Image控件的Source属性，输出结果如图29-21所示。

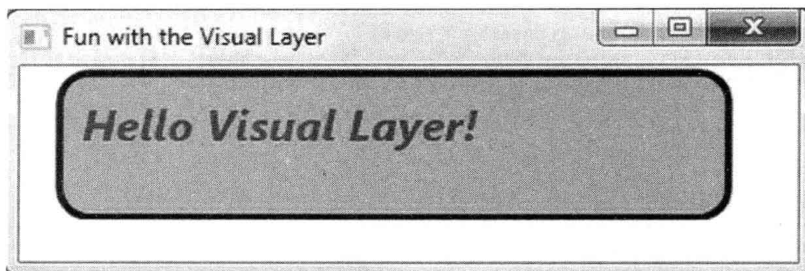


图29-21 使用可视化层呈现内存中的位图

说明 System.Windows.Media.Imaging命名空间包含许多额外的编码类，可以将内存中的Render-TargetBitmap对象保存为不同格式的物理文件。更详细的内容可以查阅JpegBitmapEncoder类（或其他编码类）。

---

3. 在自定义布局管理器中呈现可视化数据

尽管使用DrawingVisual绘制WPF控件的背景很有趣，但更常见的场景是构建自定义的布局管理器（如Grid、StackPanel、Canvas等），其内容由可视化层呈现。如果你创建了这样的自定义布局管理器，可以将其放到普通的Window（Page或UserControl）中，使UI的某个部分使用高度优化的呈现代理，而宿主窗体的非关键部分使用形状和绘图呈现剩余的图形数据。

如果你不需要布局管理器的其余功能，可以选择简单地扩展FrameworkElement，它仅包含使用可视化项必需的基础设施。为了演示这些，我们在项目中新建CustomVisualFrameworkElement类，从FrameworkElement扩展该类并导入System.Windows、System.Windows.Input和System.Windows.Media命名空间。

29

该类将维护一个VisualCollection类型的成员变量，它包含两个固定的DrawingVisual对象。（当然，你可以通过鼠标操作为该集合添加新的成员，但本例我们尽可能地保持简单。）其代码如下所示：

```
class CustomVisualFrameworkElement : FrameworkElement
{
  // 我们所创建的所有可视化对象的集合
  VisualCollection theVisuals;

  public CustomVisualFrameworkElement()
  {
    // 使用若干DrawingVisual对象填充VisualCollection
    // 构造函数的参数表示这些可视化对象的拥有者
    theVisuals = new VisualCollection(this);
    theVisuals.Add(AddRect());
    theVisuals.Add(AddCircle());
  }
  private Visual AddCircle()
```

```
  {
    DrawingVisual drawingVisual = new DrawingVisual();

    // 获取DrawingContext，创建新的绘图内容
    using (DrawingContext drawingContext = drawingVisual.RenderOpen())
    {
      // 创建一个圆圈，并在DrawingContext中进行绘制
      Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
      drawingContext.DrawEllipse(Brushes.DarkBlue, null, new Point(70, 90), 40, 50);
    }
      return drawingVisual;
  }

  private Visual AddRect()
  {
    DrawingVisual drawingVisual = new DrawingVisual();
    using (DrawingContext drawingContext = drawingVisual.RenderOpen())
    {
      Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
      drawingContext.DrawRectangle(Brushes.Tomato, null, rect);
    }
    return drawingVisual;
  }
}
```

现在，在将自定义的FrameworkElement应用于窗体之前，我们必须重写前面提到的两个关键的虚方法，它们都将在呈现过程中由WPF在内部进行调用。GetVisualChild()方法返回子元素集合中指定索引的元素。只读的VisualChildrenCount属性返回该可视化集合中可视化子元素的数量。这两个方法实现起来都非常简单，因为我们可以将实际的工作委托给VisualCollection成员变量：

```
protected override int VisualChildrenCount
{
  get { return theVisuals.Count; }
}

protected override Visual GetVisualChild(int index)
{
  // 参数值必须大于0，并且要做完整性检查
  if (index < 0 || index >= theVisuals.Count)
  {
    throw new ArgumentOutOfRangeException();
  }

  return theVisuals[index];
}
```

我们只包含了测试自定义类的最少功能。更新Window的XAML描述，在StackPanel中添加一个CustomVisualFrameworkElement对象。这需要你创建一个自定义的XML命名空间来映射.NET命名空间（参见第28章）。

```
<Window x:Class="RenderingWithVisuals.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:custom="clr-namespace:RenderingWithVisuals"
  Title="Fun with the Visual Layer" Height="350" Width="525"
  Loaded="Window_Loaded" WindowStartupLocation="CenterScreen">
    <StackPanel Background="AliceBlue" Name="myStackPanel">
        <Image Name="myImage" Height="80"/>
```

```
        <custom:CustomVisualFrameworkElement/>
    </StackPanel>
</Window>
```

如果一切正常，运行程序将看到如图29-22所示的界面。

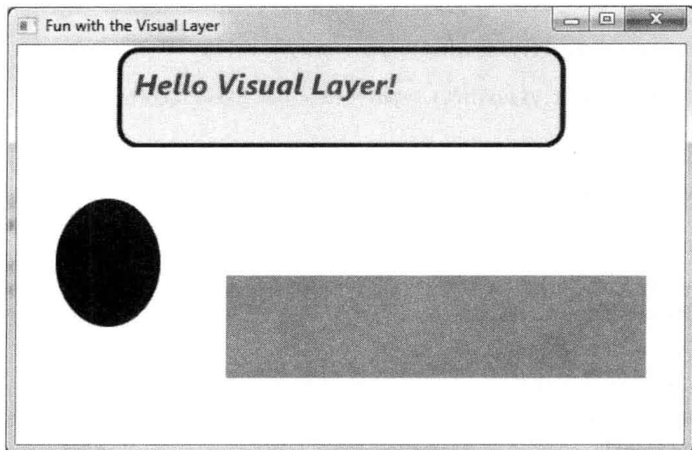


图29-22 使用可视化层在自定义的FrameworkElement中呈现数据

4. 响应命中测试操作

由于DrawingVisual不包含UIElement和FrameworkElement的任何基础结构，因此你需要手工编写程序来添加计算命中测试操作的功能。幸运的是，逻辑和可视化树的概念使得在可视化层进行这些操作变得十分简单。事实证明，当你编写XAML时，你实质上就是在构建一个元素的逻辑树。然而每个逻辑树的背后都有一个描述更加丰富的可视化树，它也包含低级别的呈现指令。

第31章将深入研究这些树的细节，不过现在我们仅需要了解，只有当你向这些数据结构注册自定义的可视化对象后，才能执行命中测试操作。幸而，VisualCollection容器已经为我们做好了这一切。(这也解释了为什么我们需要将自定义的FrameworkElement作为VisualCollection构造函数的参数。)

首先，更新CustomVisualFrameworkElement类，在构造函数中使用标准C#语法添加MouseDown事件处理程序，如下所示：

```
this.MouseDown += MyVisualHost_MouseDown;
```

该事件处理程序的实现将调用VisualTreeHelper.HitTest()方法，检查鼠标是否在所呈现的可视化对象的边界之内。我们需要为HitTest()指定一个HitTestResultCallback委托作为参数来执行这个计算。如果单击了可视化对象，我们将在它的原始呈现和扭曲呈现两者之间进行切换。为CustomVisualFrameworkElement类添加如下的方法：

```
void MyVisualHost_MouseDown(object sender, MouseButtonEventArgs e)
{
  // 找到用户单击的位置
  Point pt = e.GetPosition((UIElement)sender);

  // 通过委托来调用辅助方法，检查是否单击了可视化对象
  VisualTreeHelper.HitTest(this, null,
    new HitTestResultCallback(myCallback), new PointHitTestParameters(pt));
```

29

```
}

public HitTestResultBehavior myCallback(HitTestResult result)
{
  // 如果单击了可视化对象，则在其原始呈现和扭曲呈现之间进行切换
  if (result.VisualHit.GetType() == typeof(DrawingVisual))
  {
    if (((DrawingVisual)result.VisualHit).Transform == null)
    {
      ((DrawingVisual)result.VisualHit).Transform = new SkewTransform(7, 7);
    }
    else
    {
      ((DrawingVisual)result.VisualHit).Transform = null;
    }
  }

  // 告诉HitTest()停止深入可视化树
  return HitTestResultBehavior.Stop;
}
```

再次运行程序，现在可以单击所呈现的可视化对象并看到它的变换行为。这只是关于WPF可视化层非常简单的一个示例，你还可以在XAML中使用同样的画笔、画刷、图形变换、布局管理器等。这样，你就已经对如何使用Visual派生类有了相当多的了解。

源代码　RenderingWithVisuals项目的源代码位于Chapter 29子目录下。

这就是我们要介绍的WPF图形呈现服务的所有内容。尽管我们涵盖了很多有趣的话题，但实际上只涉及了WPF图形功能的皮毛。我将深入研究形状、绘图、画刷、图形变换和可视化等话题的事情留给读者。（当然，可以肯定的是，你将在其余WPF章节中看到这些话题的更多细节。）

29.9　小结

由于WPF是一个图形密集的GUI API，因此自然而然地提供了多种呈现图形输出的方式。本章开头分别介绍了3种WPF应用程序呈现图形的方式（形状、绘图和可视化），接着讨论了不同的呈现基元，如画刷、画笔和图形变换。

在构建交互式的2D呈现时，形状是最简单的方法。但对于静态的、非交互式的呈现来说，使用绘图和几何图形则是最佳的方式，而可视化层（只能用代码访问）提供了更多的控制和更优的性能。

第 30 章 WPF资源、动画和样式

本章介绍三个重要的（也是互相关联的）话题，它们将加深你对WPF API的理解。首先要学习的是逻辑资源的作用。稍后你会看到，逻辑资源（也称为对象资源）系统是一种在WPF应用程序中对常用对象进行命名和引用的方式。逻辑资源常常存在于XAML中，不过它们也可以定义在程序代码中。

接下来你将学习如何定义、执行和控制一个动画序列。也许会出乎你的意料，WPF动画可不仅局限于视频游戏和多媒体应用。在WPF API下，动画可以使按钮在被单击后神奇地发光，或使DataGrid的选中行变得很大。理解动画是构建自定义控制模板（将在第31章中介绍）的关键。

最后我们将探索WPF样式的作用。WPF应用程序可以为控件定义通用的外观和体验，这与网页中使用CSS或ASP.NET主题引擎非常类似。你可以将这些样式定义在标记中，并保存为对象资源供以后使用，也可以在运行时动态地加载它们。

30.1 理解 WPF 资源系统

我们的第一个任务是研究嵌入和访问应用程序资源这个话题。WPF支持两种风格的资源。第一种是二进制资源（binary resource），这也是大多数程序员在传统意义上所理解的那种资源（如内嵌的图像文件或声音片段、应用程序图标，等等）。

第二种称为对象资源（object resource）或逻辑资源（logical resource），它表示一个命名的.NET对象，可以打包并在整个应用程序中进行复用。由于所有.NET对象都可以打包为对象资源，因此你可以定义通用的图像基元（如画刷、画笔、动画等），并在需要时进行引用，这在处理各种图形数据时是非常有帮助的。

使用二进制资源

在展开对象资源的话题之前，我们先来快速学习一下如何打包诸如图标或图像文件（如公司logo或动画中的图片）这种二进制资源。用Visual Studio创建一个新的WPF应用程序BinaryResourcesApp。修改初始的窗体标记，用DockPanel作为根布局：

```
<Window x:Class="BinaryResourcesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Binary Resources" Height="500" Width="649">

  <DockPanel LastChildFill="True">
```

```
  </DockPanel>
</Window>
```

现在，假设你的应用程序会根据用户的输入显示窗体内部三个图像文件中的一个。WPF Image控件不仅可以显示典型的图像文件（*.bmp、*.gif、*.ico、*.jpg、*.png、*.wdp和*.tiff），还可以显示DrawingImage中的数据（参见第29章）。我们来为窗体建立一个UI，其DockPanel中有一个工具条，工具条包含Next和Previous按钮。在工具条下方放置一个Image控件，目前没有设置其Source属性的值，因为我们要在代码中进行操作：

```
<Window x:Class="BinaryResourcesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Binary Resources" Height="500" Width="649">

  <DockPanel LastChildFill="True">
    <ToolBar Height="60" Name="picturePickerToolbar" DockPanel.Dock="Top">
      <Button x:Name="btnPreviousImage" Height="40" Width="100" BorderBrush="Black"
              Margin="5" Content="Previous" Click="btnPreviousImage_Click"/>
      <Button x:Name="btnNextImage" Height="40" Width="100" BorderBrush="Black"
              Margin="5" Content="Next" Click="btnNextImage_Click"/>
    </ToolBar>

    <!-- 我们将在代码中填充Image -->
    <Border BorderThickness="2" BorderBrush="Green">
      <Image x:Name="imageHolder" Stretch="Fill" />
    </Border>
  </DockPanel>
</Window>
```

注意，每个Button对象都对Click事件进行了处理。假设已经使用IDE处理了这些事件，那么你将在C#代码中得到两个空方法。那么，我们如何编写Click事件处理程序来循环图像数据呢？更重要的是，我们是希望图像数据位于用户硬盘还是嵌入到编译的程序集中？让我们来看看我们的选择。

1. 在项目中使用松散的资源文件

假设你希望将图像作为松散文件发布在应用程序安装路径的某个子目录中。在Visual Studio的Solution Explorer窗体中，右击项目节点并选择Add→New Folder菜单选项创建一个子目录Images。

现在，右击该文件夹，选择Add→Existing Item菜单选项，将图像文件复制到子目录中。在本项目的可下载源代码中，你会发现3个图像文件Welcome.jpg、Dogs.jpg和Deer.jpg。你可以将它们添加到项目中，也可以自行选择3个图像文件。图30-1显示了当前的设置。

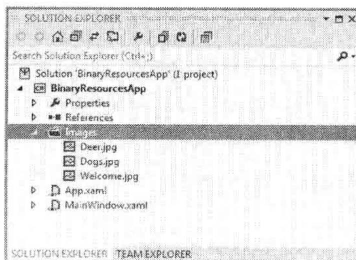


图30-1　WPF项目中包含图像数据的子目录

2. 配置松散资源

当使用Visual Studio向输出目录复制项目内容时，你需要在Properties窗体中调整一些设置。要确保\Images文件夹下的内容全部复制到\bin\Debug目录下，首先选择Solution Explorer中的所有文件。然后，在Properties窗体中，将Build Action属性设置为Resource，将Copy to Output Directory属性设置为Copy always（如图30-2所示）。

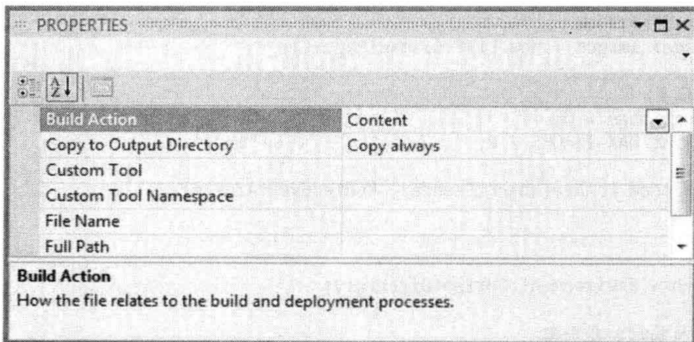


图30-2　设置图像数据使其能够复制到输出目录中

重新编译程序，单击Solution Explorer中的Show all Files按钮，查看\bin\Debug目录下的Image文件夹（可能需要单击刷新按钮）。如图30-3所示。

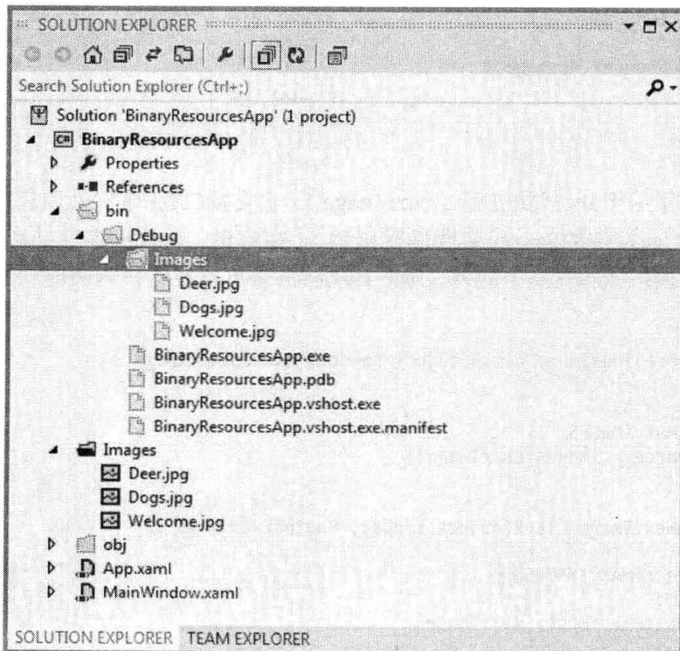


图30-3　复制的数据

3. 加载图像

WPF提供了一个BitmapImage类，位于System.Windows.Media.Imaging命名空间下。它可以加载一个位置由System.Uri对象表示的图像文件的数据。你可以向下面这样在窗体的Loaded事件处理程序中填充BitmapImage类型的List<T>：

```
public partial class MainWindow : Window
{
  // BitmapImage文件的列表
  List<BitmapImage> images = new List<BitmapImage>();

  // 列表的当前位置
  private int currImage = 0;
  private const int MAX_IMAGES = 2;

  private void Window_Loaded(object sender, RoutedEventArgs e)
  {
    try
    {
      string path = Environment.CurrentDirectory;

      // 在窗体加载时加载图像
      images.Add(new BitmapImage(new Uri(string.Format(@"{0}\Images\Deer.jpg", path))));
      images.Add(new BitmapImage(new Uri(string.Format(@"{0}\Images\Dogs.jpg", path))));
      images.Add(new BitmapImage(new Uri(string.Format(@"{0}\Images\Welcome.jpg", path))));

      // 显示列表中的第一幅图像
      imageHolder.Source = images[currImage];
    }
    catch (Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
  }
...
}
```

注意该类还定义了一个int成员变量（currImage），它支持Click事件处理程序遍历List<T>中的各项，并将其赋值给Image控件的Source属性进行显示。（在这里，Loaded事件处理程序将Source属性的值设置为List<T>中的第一幅图像。）此外，MAX_IMAGES常量可以在迭代列表时检查其上下限。Click事件处理程序如下所示：

```
private void btnPreviousImage_Click(object sender, RoutedEventArgs e)
{
  if (--currImage < 0)
    currImage = MAX_IMAGES;
  imageHolder.Source = images[currImage];
}

private void btnNextImage_Click(object sender, RoutedEventArgs e)
{
  if (++currImage > MAX_IMAGES)
    currImage = 0;
  imageHolder.Source = images[currImage];
}
```

运行程序，就可以翻阅每张图片了。

4. 内嵌应用程序资源

如果你希望将图像文件作为二进制资源直接编译到.NET程序集中，需要在Solution Explorer中选择图像文件（在\Images目录下，而非\bin\Debug\Images目录）。然后将Build Action属性改为Resource、Copy to Output Directory属性改为Do not copy（如图30-4所示）。

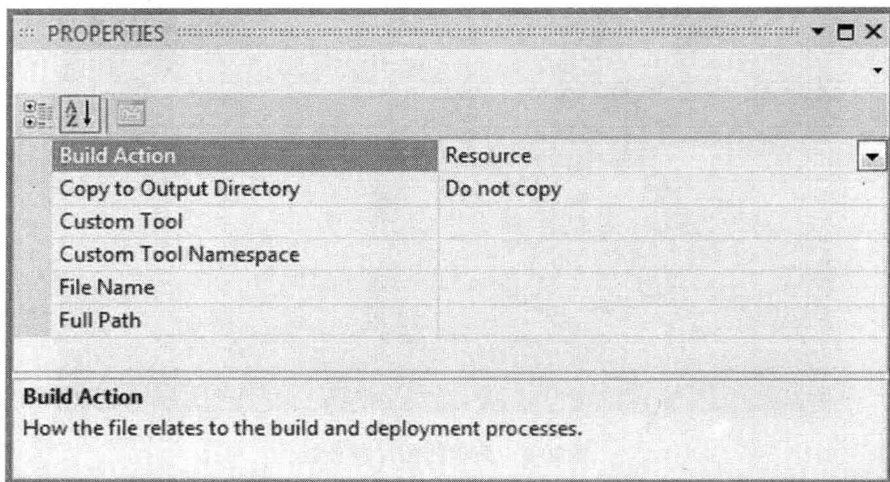


图30-4　将图像配置为内嵌资源

现在打开Visual Studio的Build菜单，选择Clean Solution选项清空\bin\Debug\Images下的当前内容，然后重新编译项目。刷新Solution Explorer，观察\bin\Debug目录下是否还存在Images子目录。使用当前的生成选项，图像数据将不会复制到输出目录，而是内嵌到程序集中。

这样调整之后，我们需要修改代码，提取编译到程序集中的图像进行加载：

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  try
  {
    images.Add(new BitmapImage(new Uri(@"/Images/Deer.jpg", UriKind.Relative)));
    images.Add(new BitmapImage(new Uri(@"/Images/Dogs.jpg", UriKind.Relative)));
    images.Add(new BitmapImage(new Uri(@"/Images/Welcome.jpg", UriKind.Relative)));
    imageHolder.Source = images[currImage];
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

注意，在本例中我们不再需要指定安装路径，只需要简单地列出资源名称及其所在子目录的名称。此外，在创建Uri对象时，我们还将UriKind指定为Relative。这样，无论在什么情况下我们的可执行文件都是一个独立的实体，可以在电脑的任何路径上运行，因为所有的编译数据都已经存在于二进制文件中。图30-5显示了完整的应用程序。

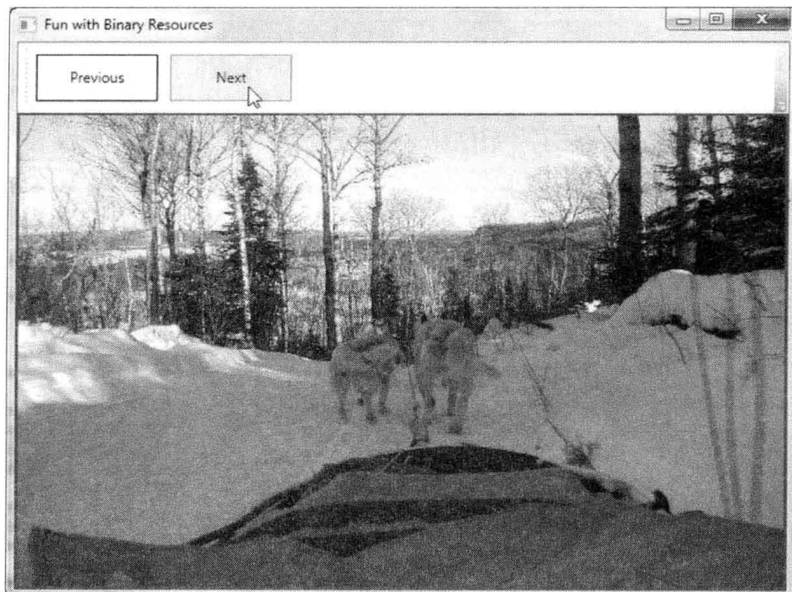


图30-5　简单的图片查看器

源代码　BinaryResourcesApp的源代码位于Chapter 30子目录下。

30.2　使用对象（逻辑）资源

在构建WPF应用时，我们常常定义一段XAML，将其用于一个窗体的多个位置，或横跨多个窗体或项目。例如，假设你正在使用Expression Blend创建完美的线性渐变画刷，包含10行标记。现在你希望将该画刷作为项目中所有Button的背景，该项目包含8个窗体，16个Button。

将XAML复制粘贴到每个控件是最糟糕的做法。在你需要调整画刷的外观时，这无疑将是一场噩梦。

谢天谢地，对象资源可以定义一小段命名的XAML，并将其保存在合适的字典中供以后使用。与二进制资源类似，对象资源通常都编译到需要它们的程序集中。不过你不需要修改Build Action属性。我们只需将XAML放置到正确的位置，编译器将完成剩下的工作。

使用对象资源是WPF开发的重要部分。正如你将看到的那样，对象资源比自定义画刷要复杂得多。你可以定义基于XAML的动画、3D呈现、自定义控件样式、数据模板、控件模板，等等，并将它们统统打包成可重用的资源。

30.2.1　Resources属性的作用

如前所述，要在应用程序中使用对象资源，必须将它们放置在合适的字典对象中。每个FrameworkElement派生类都包含一个Resources属性，它封装了一个ResourceDictionary对象，用来保存定义

的对象资源。由于ResourceDictionary基于System.Object类型并且可由XAML或程序代码进行操作，因此可以保存任何类型。

在WPF中，所有控件、Window、Page（在创建导航应用或XBAP程序时使用）和UserControl都扩展了FrameworkElement，因此所有部件都可以访问ResourceDictionary。此外，尽管Application类未继承FrameworkElement，但它也支持一个同名的Resources属性并且作用也相同。

30.2.2 定义窗口级别的资源

在开始探索对象资源的作用之前，先使用Visual Studio新建一个WPF应用程序ObjectResources-App，并将初始的Grid修改为水平对齐的StatckPanel布局管理器。在这个StackPanel中定义两个Button控件（我们不需要太多的东西来演示对象资源的作用，这些就足够了）：

```
<Window x:Class="ObjectResourcesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Object Resources" Height="350" Width="525">

  <StackPanel Orientation="Horizontal">
    <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"/>
    <Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
  </StackPanel>

</Window>
```

现在，使用集成的画刷编辑器（在第29章中讨论过）将OK按钮的Background颜色属性设置为自定义的画刷类型。注意画刷是如何嵌入到<Button>和</Button>标签作用域内的：

```
<Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20">
  <Button.Background>
    <RadialGradientBrush>
      <GradientStop Color="#FFC44EC4" Offset="0" />
      <GradientStop Color="#FF829CEB" Offset="1" />
      <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
  </Button.Background>
</Button>
```

要想让Cancel按钮也能使用该画刷，我们需要将<RadialGradientBrush>的作用域扩大到父元素的资源字典中。例如，将其移动到<StatckPanel>中，由于两个按钮都位于该布局管理器中，因此可以使用同样的画刷。我们甚至可以将画刷打包到window本身的资源字典中，这样窗体内容中（如内嵌的面板）的所有部分都可以随意使用。

在定义资源时，可以使用属性元素语法设置Resources属性。你还需要给资源项设置一个x:Key值，当窗体的其他部分希望引用该对象资源时，可以使用它。要注意x:Key和x:Name是不同的！x:Name特性可以将对象作为代码文件中的成员变量进行访问，而x:Key特性用来引用资源字典中的某项。

Visual Studio和Expression Blend都可以使用相应的Properties窗口将资源提升到一个更高的作用域。在Visual Studio中，首先确定类型为复杂对象并希望将其封装为资源的属性（本例中的Background属性）。在该属性旁边可以看到一个白色小方形按钮，单击可以打开一个弹出菜单。在弹出菜单中选择“Convert to New Resource...”选项（如图30-6所示）。

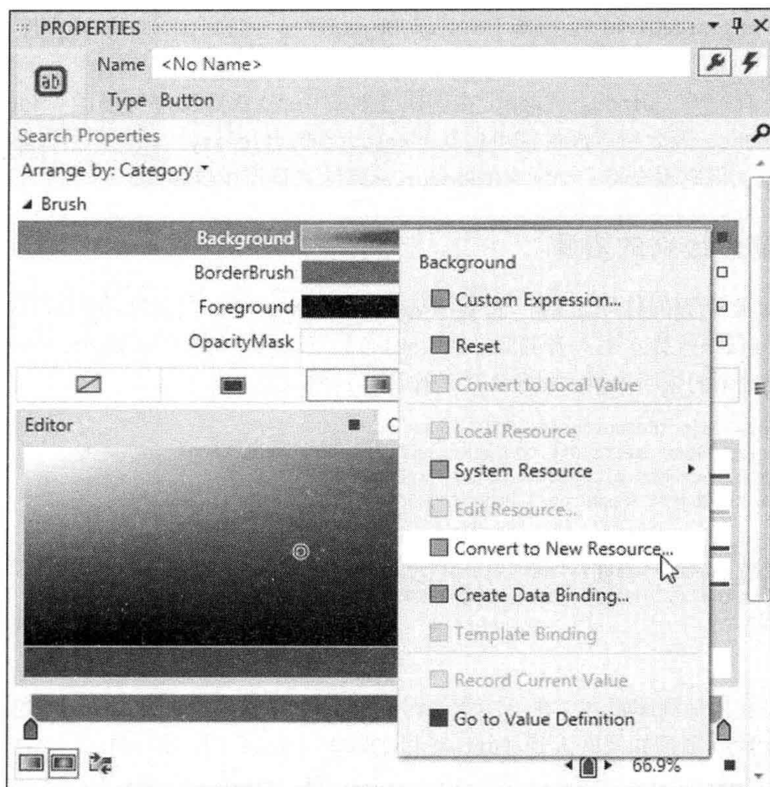


图30-6　将复杂对象移至资源容器

然后将要求你对资源命名（myBrush），并指定其位置。本例中保留当前文档的默认选择（如图30-7所示）。

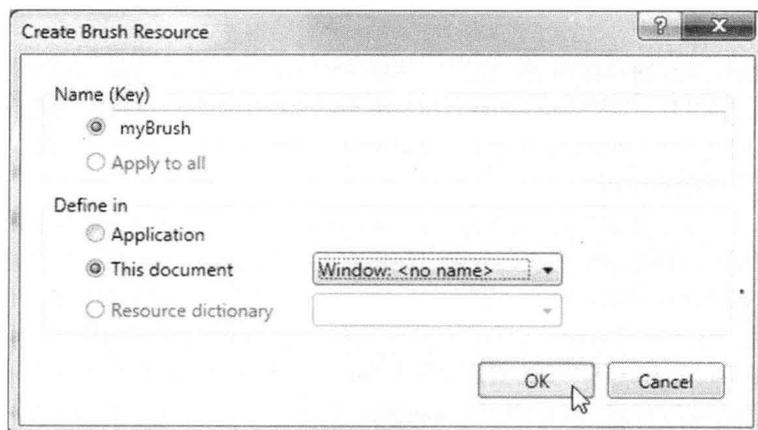


图30-7　命名对象资源

完成之后，标记将被调整为如下形式：

```
<Window x:Class="ObjectResourcesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Object Resources" Height="350" Width="525">

  <Window.Resources>
    <RadialGradientBrush x:Key="myBrush">
      <GradientStop Color="#FFC44EC4" Offset="0" />
      <GradientStop Color="#FF829CEB" Offset="1" />
      <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
  </Window.Resources>

  <StackPanel Orientation="Horizontal">
    <Button Margin="25" Height="200" Width="200" Content="OK"
            FontSize="20" Background="{StaticResource myBrush}"></Button>
    <Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
  </StackPanel>

</Window>
```

注意，新的<Window.Resources>标签包含一个RadialGradientBrush对象，其键值为myBrush。

30.2.3 {StaticResource}标记扩展

提取对象资源后发生的另一个变化是，提取的目标属性（还是Background）现在使用了{StaticResource}标记扩展，并用键的名称作为参数。现在，Cancel按钮可以随意使用同样的画刷来绘制背景。或者，如果Cancel按钮包含某些复杂内容，Button的任何子元素都可以使用这个窗体级别的资源——例如，Ellipse的Fill属性：

```
<StackPanel Orientation="Horizontal">
  <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"
          Background="{StaticResource myBrush}">
  </Button>

  <Button Margin="25" Height="200" Width="200" FontSize="20">
    <StackPanel>
      <Label HorizontalAlignment="Center" Content= "No Way!"/>
      <Ellipse Height="100" Width="100" Fill="{StaticResource myBrush}"/>
    </StackPanel>
  </Button>
</StackPanel>
```


30.2.4 {DynamicResource}标记扩展

当连接资源时，属性也可以使用{DynamicResource}扩展标记。要了解其不同之处，先将OK按钮命名为btnOK，并添加Click事件处理程序。在事件处理程序中使用Resources属性获取自定义画刷，并对其进行修改：

```
private void btnOK_Click(object sender, RoutedEventArgs e)
{
  // 获取画刷并进行修改
  RadialGradientBrush b = (RadialGradientBrush)Resources["myBrush"];
```

```
    b.GradientStops[1] = new GradientStop(Colors.Black, 0.0);
  }
```

说明　这里我们使用Resources索引器通过名称来定位资源。但是，要知道如果没有找到资源将会抛出一个运行时异常。你还可以使用TryFindResource()方法，在制定的资源不存在的情况下返回null，而不会抛出运行时错误。

运行程序并单击OK按钮，你会发现画刷发生了改变，并且所有的按钮也都用修改后的画刷进行了更新。但是，如果完全更改由myBrush键指定的画刷类型会发生什么样的情况呢？例如：

```
private void btnOK_Click(object sender, RoutedEventArgs e)
{
  // 用全新的画刷替换原来的myBrush
  Resources["myBrush"] = new SolidColorBrush(Colors.Red);
}
```

这时再单击按钮，不会发生任何改变。这是因为{StaticResource}扩展标记只加载一次资源，并在应用程序声明周期与原始对象保持“连接”。不过，如果我们将所有{StaticResource}改为{DynamicResource}，那么我们的自定义画刷将会如期地变为纯红色的画刷。

实质上，{DynamicResource}扩展标记可以探测实际的对象是否被新的对象所替换。正如你猜测的那样，这需要额外的运行时基础结构，因此除非你确定会在运行时更改对象资源，并且希望通知所有使用该资源的项，否则请坚持使用{StaticResource}。

30.2.5　应用程序级别的资源

当窗体的资源字典有了对象资源以后，窗体中的所有项都可以自由使用这些资源，但同一个应用程序中的其他窗体却无法享用。将Cancel按钮命名为btnCancel，并添加Click事件处理程序。为当前项目添加包含一个按钮的新窗体（命名为TestWindow.xaml），单击按钮将关闭该窗体：

```
public partial class TestWindow : Window
{
  public TestWindow()
  {
    InitializeComponent();
  }

  private void btnClose_Click(object sender, RoutedEventArgs e)
  {
    this.Close();
  }
}
```

在第一个窗体Cancel按钮的Click事件处理程序中，加载并显示这个新的window，如下：

```
private void btnCancel_Click(object sender, RoutedEventArgs e)
{
  TestWindow w = new TestWindow();
  w.Owner = this;
  w.WindowStartupLocation = WindowStartupLocation.CenterOwner;
  w.ShowDialog();
}
```

新的窗体无法使用myBrush，因为它不在同一个“作用域”之内。解决方法是在应用程序级别定义对象资源，而不是某个特定的窗体级别。不过这无法在Visual Studio中自动完成，所以需要将当前画刷对象从<Windows.Resources>作用域剪切出来，复制到App.xaml文件的<Application.Resources>作用域中：

```
<Application x:Class="ObjectResourcesApp.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">

  <Application.Resources>
    <RadialGradientBrush x:Key="myBrush">
      <GradientStop Color="#FFC44EC4" Offset="0" />
      <GradientStop Color="#FF829CEB" Offset="1" />
      <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
  </Application.Resources>

</Application>
```

现在TestWindow可以随意使用这个画刷来绘制背景了。找到新Window的Background属性，单击Brush resources选项卡查看应用程序级别的资源（如图30-8所示）。

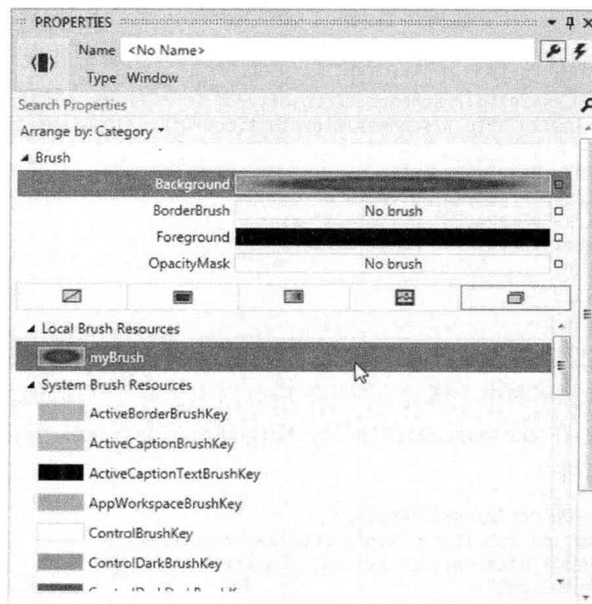


图30-8 使用应用程序级别的资源

30.2.6 定义合并的资源字典

应用程序级别的资源是个不错的选择，但如果你希望定义一些复杂（或不那么复杂）的资源并在多个WPF项目中复用，应该如何是好呢？这时你需要定义合并的资源字典。它只不过是一个包含了对象资源集合的.xaml文件。单个项目可以包含任意多个这种文件（用来保存所有画刷的文件、用

来保存所有动画的文件，等等），每个文件都能通过Project菜单的Add New Item对话框插入（如图30-9所示）。

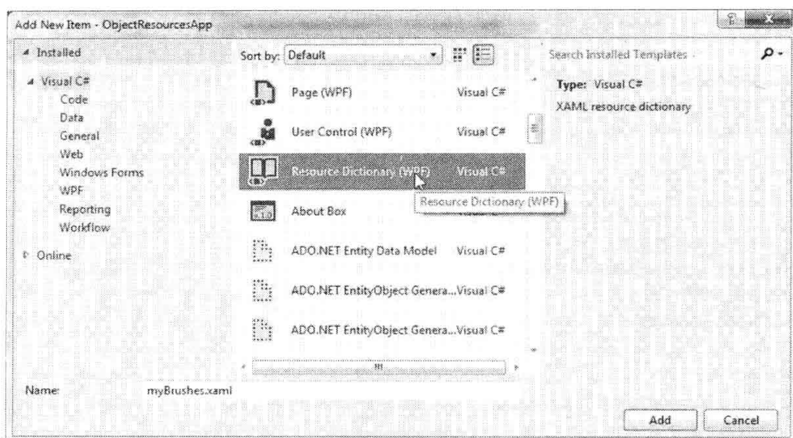


图30-9　插入新的合并的资源字典

在MyBrushes.xaml文件中，我们将Application.Resources作用域中的资源转移到字典中，如下：

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44EC4" Offset="0" />
    <GradientStop Color="#FF829CEB" Offset="1" />
    <GradientStop Color="#FF793879" Offset="0.669" />
  </RadialGradientBrush>

</ResourceDictionary>
```

现在，尽管资源字典已经成为项目的一部分，但它仍然将产生运行时错误。因为所有资源字典都必须合并到一个已知的资源字典中（通常为应用程序级别的字典）。可以使用如下的格式（注意，多个资源字典可以通过在<ResourceDictionary.MergedDictionaries>作用域内添加<ResourceDictionary>元素来进行合并）。

```
<Application x:Class="ObjectResourcesApp.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">

  <!-- 引入MyBrushes.xaml文件中的逻辑资源-->
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source = "MyBrushes.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>

</Application>
```

30.2.7 定义只含资源的程序集

最后，你可以创建只包含对象资源字典的.NET类库。定义可用于计算机级别的主题是非常有用的。你可以将对象资源打包到一个辅助的程序集中，要用到它们的应用程序可以将其加载到内存中。

要构建一个只含资源的程序集，最简单的方式是通过WPF User Control Library项目。在当前解决方案中添加这样一个项目（取名为MyBrushesLibrary），使用Visual Studio中的Add→New Project菜单项（如图30-10所示）。

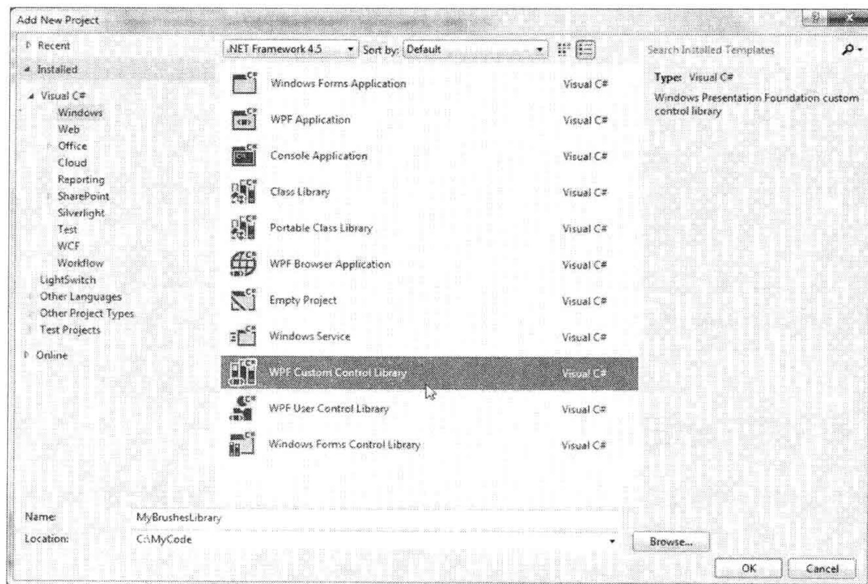


图30-10 为构建只含资源的程序集添加一个User Control Library

然后，彻底删除项目中的UserControl1.xaml（我们只需要被引用的WPF程序集）。接下来，将MyBrushes.xaml文件拖曳到MyBrushes Library项目中，并从ObjectResourcesApp项目中删除。Solution Explorer如图30-11所示。

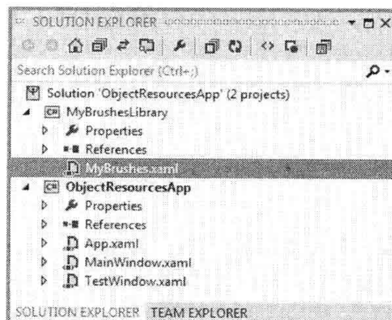


图30-11 将MyBrushes.xaml文件移动到新的库项目中

编译User Control Library项目。然后使用Add Reference对话框在ObjectResourcesApp项目中引用该库。现在，你可以在ObjectResourcesApp项目中的应用程序级别的资源字典中合并这些二进制资源。合并的语法有点晦涩，如下：

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <!-- 语法为/NameOfAssembly;Component/NameOfXamlFileInAssembly.xaml -->
      <ResourceDictionary Source = "/MyBrushesLibrary;Component/MyBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

首先，要注意这个字符串对空格敏感。如果在分号或斜线前后添加了空格，将得到运行时错误。该字符串的第一部分是外部库的名称（不含文件扩展名）。在分号之后，输入单词Component，然后是编译的二进制资源的名称，与原始的XAML资源字典相同。

这就是我们对于WPF资源管理系统的研究。你可以在大多数应用程序以及本书其他WPF章节中好好使用这些技术。接下来，我们开始研究WPF中集成的动画API。

源代码　ObjectResourcesApp项目的源代码位于Chapter 30子目录下。

30.3　理解WPF动画服务

除了第29章介绍的图形呈现服务以外，WPF还提供了支持动画服务的编程接口。动画这个词意味着炫动的公司logo、旋转的图像资源序列（为了产生移动的视觉效果）、在屏幕上跳动的文字或视频游戏、多媒体应用这样的特殊类型的应用程序。

WPF动画API完全能够达到这些目的，并且还能随时为应用程序添加额外的效果。例如，你可以为按钮添加一个动画效果，当鼠标光标悬浮在按钮边框时按钮会稍稍放大（当鼠标光标移开边框时，按钮将恢复原来的大小）。或者为窗体添加一个动画，使其在关闭时产生特殊的视觉效果，如慢慢变为透明。事实上，如果你希望提供更丰富的用户体验，可以将WPF动画用于各种应用程序（商业应用、多媒体程序、视频游戏，等等）。

如同WPF的其他方面一样，构建动画的概念也不是全新的。与以前用到的那些API（如Windows Form）所不同的是，你不需要手动编写底层代码。使用WPF，你不需要创建后台线程或定时器以推进动画序列，不需要定义自定义类型来表示动画，不需要擦除和重绘图像以及其他烦琐的数学计算。

和WPF的其他方面类似，你可以完全使用XAML构建动画，也可以完全使用C#代码，或者两者相结合。

说明　Visual Studio没有提供创建动画的GUI动画工具。如果希望使用Visual Studio创建动画，需要直接输入XAML。实际上Expression Blend有个内置的动画编辑器，可以大大简化相关操作。

30.3.1 动画类型的作用

要想理解WPF对于动画的支持，必须先研究PresentationCore.dll中System.Windows.Media.Animation命名空间下的各种动画类。使用Animation标记命名的类型超过了100个。

这些类型可以分为三大类。首先，遵循*DataType*Animation命名约定的类（ByteAnimation、ColorAnimation、DoubleAnimation、Int32Animation等）可以使用线性插值动画。你可以将某个值从起点到终点随着时间的推移缓慢地变化。

其次，遵循*DataType*AnimationUsingKeyFrames命名约定的类（StringAnimationUsingKeyFrames、DoubleAnimationUsingKeyFrames、PointAnimationUsingKeyFrames等）表示"关键帧动画"，可以在一段时间内循环一组已定义的值。例如，你可以使用关键帧通过循环一系列单个字符来改变按钮的标题。

最后，遵循*DataType*AnimationUsingPath命名约定的类（DoubleAnimationUsingPath、PointAnimationUsingPath等）是基于路径的动画，它可以使对象按照定义好的路径移动。举个例子，如果你正在构建一个GPS应用程序，就可以使用基于路径的动画，使某个东西按照最快路径到达用户的目的地。

显然，这些类不是用来直接为某个特殊的数据类型变量提供动画序列的。（那么，我们究竟如何才能使用Int32Animation对"9"这个值执行动画呢？）

例如，Label类型的Height和Width属性都是double类型的依赖属性。如果你希望定义一个动画在某个时间跨度内增加标签的高度，可以将一个DoubleAnimation对象关联到Height属性，并允许WPF处理执行实际动画的细节。再举一个例子，如果你希望在5秒钟内将一个画刷类型的颜色由绿色改为黄色，可以使用ColorAnimation类型。

这些Animation类都可以关联到类型匹配的*依赖属性*。正如第31章将介绍的那样，依赖属性是特殊类型的属性，很多WPF服务如动画、数据绑定和样式都要用到。

按照惯例，依赖属性定义为类的静态只读字段，命名方式为普通属性名称加上Property后缀。例如，Button的Height属性的依赖属性为Button.HeightProperty。

30.3.2 To、From 和 By 属性

所有Animation类都定义了几个关键属性用来控制执行动画的开始和结束值。

- To：表示动画的结束值。
- From：表示动画的起始值。
- By：标识要改变的起始值的总数目。

尽管所有的Animation类都支持To、From和By属性，但它们并未继承基类的虚成员。理由是这些属性所封装的基本类型不甚相同（如整型、颜色、Thickness对象等），使用一个基类来表示这些属性将导致复杂的代码结构。

你可能也不知道为什么没有使用.NET泛型来定义一个包含单个类型参数的泛型动画类（如Animation<T>）。同样，由于大量的基本数据类型（颜色、向量、整数、字符串等）用于动画的依赖属性，使用泛型不会得到一个预期的干净的解决方案（更何况XAML对泛型类型的支持也很有限）。

30.3.3　Timeline基类的作用

尽管没有使用单独的基类定义To、From和By虚属性，这些Animation类却都继承了基类System.Windows.Media.Timeline。该类型提供了大量的属性来控制动画的进度，如表30-1所示。

表30-1　Timeline基类的关键成员

属　性	含　义
AccelerationRatio、DecelerationRatio、SpeedRatio	这些属性可以用来控制动画序列的整体进度
AutoReverse	获取或设置一个值，指示时间段在完成向前的循环后是否自动回放（默认值为false）
BeginTime	获取或设置时间段开始的时间。默认值为0，表示立即开始动画
Duration	设置播放时间段的一个持续的时间
FillBehavior、RepeatBehavior	这些属性用来控制时间段完成之后要发生的事情(重复动画或什么都不做)

30.3.4　用C#代码创建动画

具体来说，我们将创建含有一个Button的窗体，每当鼠标滑入按钮表面时窗体将神奇地开始转圈（围绕左上角）。使用Visual Studio新建一个WPF应用程序SpinningButtonAnimationApp。按如下的代码修改初始的标记（注意我们处理了按钮的MouseEnter事件）：

```
<Window x:Class="SpinningButtonAnimationApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Animations in C# code" Height="350"
  Width="525" WindowStartupLocation="CenterScreen">
  <Grid>
    <Button x:Name="btnSpinner" Height="50" Width="100" Content="I Spin!"
            MouseEnter="btnSpinner_MouseEnter"/>
  </Grid>
</Window>
```

现在，在C#代码文件中引入System.Windows.Media.Animation命名空间并添加如下代码：

```
public partial class MainWindow : Window
{
  private bool isSpinning = false;

  private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
  {
    if (!isSpinning)
    {
      isSpinning = true;

      // 创建一个double动画对象，并注册其Completed事件
      DoubleAnimation dblAnim = new DoubleAnimation();
      dblAnim.Completed += (o, s) => { isSpinning = false; };

      // 设置起始和结束值
      dblAnim.From = 0;
      dblAnim.To = 360;
```

```
      // 创建RotateTransform对象，并设置按钮的RenderTransform属性
      RotateTransform rt = new RotateTransform();
      btnSpinner.RenderTransform = rt;

      // 现在，执行RotateTransform对象
      rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
    }
  }
}
```

该方法的首要任务是配置DoubleAnimation对象，它的起始值为0，结束值为360。我们还处理了该对象的Completed事件，如果当前动画执行完毕，就将类级别的bool变量设置为false，让它不再重新开始。

接下来，我们创建一个RotateTransform对象，将它连接到Button控件（btnSpinner）的RenderTransform属性。最后，通知RenderTransform对象使用我们的DoubleAnimation对象对Angle属性执行动画。在编写动画代码时，通常要调用BeginAnimation()方法，将希望执行动画的依赖属性（依惯例应为类的静态字段）和相关的动画对象作为参数传入。

让我们为程序添加另一个动画，当单击按钮时，它将逐渐消失。首先，处理btnSpinner对象的Click事件，然后为事件处理程序添加如下代码：

```
private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
  DoubleAnimation dblAnim = new DoubleAnimation();
  dblAnim.From = 1.0;
  dblAnim.To = 0.0;
  btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

这里，我们改变Opacity属性的值，使按钮从视图中消失。但是现在这样做有点困难，因为按钮正在快速旋转！那么，如何控制动画的速度呢？你能这么问让我甚感欣慰。

30.3.5 控制动画的速度

默认情况下，一个动画从From的值转换到To的值大约需要1秒的时间。因此，单击按钮时它首先用1秒的时间旋转360° 然后才会逐渐消失。

如果你需要自定义动画执行的时间，可以通过动画对象的Duration属性，该属性为一个Duration对象的实例。通常情况下，将一个TimeSpan对象传入Duration的构造函数，以此来建立时间跨度。下面的代码使按钮的旋转时间达到4秒：

```
private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
{
  if (!isSpinning)
  {
    isSpinning = true;

    // 创建一个double动画对象，并注册其Completed事件
    DoubleAnimation dblAnim = new DoubleAnimation();
    dblAnim.Completed += (o, s) => { isSpinning = false; };

    // 按钮将旋转4秒
    dblAnim.Duration = new Duration(TimeSpan.FromSeconds(4));
...
```

```
    }
  }
```

经过这一调整，你将有机会在按钮旋转的时候单击它，从而使它逐渐消失。

说明 一个Animation类的BeginTime属性同样含有TimeSpan对象。该属性可以在一个动画序列开始之前建立一个等待时间。

30.3.6 动画的反转和循环

你可以将Animation对象的AutoReverse属性设置为true，这样在动画序列完成之后将自动反转动画。例如，如果希望按钮消失之后再恢复显示，可以编写如下的代码：

```
private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
  DoubleAnimation dblAnim = new DoubleAnimation();
  dblAnim.From = 1.0;
  dblAnim.To = 0.0;

  // 结束之后反转
  dblAnim.AutoReverse = true;
  btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

如果希望动画重复播放数次（或一直重复播放），可以使用所有Animation类都拥有的Repeat-Behavior属性。将一个简单的数值传入它的构造函数，就可以以硬编码的方式指定重复播放的次数。

另一方面，如果将一个TimeSpan传入构造函数，可以指定动画重复播放的持续时间。最后，如果希望动画无止境地循环下去，可以简单地指定为RepeatBehavior.Forever。在本例中，我们可以使用以下3种方法改变DoubleAnimation对象的重复行为：

```
// 一直循环
dblAnim.RepeatBehavior = RepeatBehavior.Forever;

// 循环3次
dblAnim.RepeatBehavior = new RepeatBehavior(3);

// 循环30秒
dblAnim.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(30));
```

使用C#代码和WPF动画API对对象的某些部分执行动画的研究就到此结束了。接下来我们将学习如何使用XAML做同样的事情。

源代码 SpinningButtonAnimationApp项目的源代码位于Chapter 30子目录下。

30.4 用 XAML 创建动画

用标记创建动画跟在代码中创建是类似的，至少对于简单的动画序列来说是这样。但是如果需要得到更复杂的动画，就要同时修改多个属性的值，标记数量将会显著增加。即便是使用工具来生成基于XAML的动画，了解如何使用XAML表示动画仍然是十分重要的，它可以使你更方便地调整和修改工具生成的内容。

说明 在可下载源代码的XamlAnimations文件夹中，你将发现大量的XAML文件。在进行本节剩余部分的学习时，可以将这些标记复制到自定义的XAML编辑器或Kaxaml编辑器中，并查看结果。

在标记中创建动画和用代码创建在很大程度上是一样的。你都需要配置一个Animation对象，并将它关联到某个属性。但最大的不同是，WPF对函数调用来说是不友好的。你不能调用BeginAnimation()，只能使用演示图板（storyboard）作为间接层。

让我们通过一个完整的示例来演示如何用XAML定义动画，并进行详细的说明。下面的XAML定义显示了一个含有标签的窗体。在Label对象加载到内存时，开始执行一个动画序列，在4秒之内使字体从12增大到100。只要Window对象还在内存中，这个动画就会一直重复播放。这段标记位于GrowLabelFont.xaml文件中，你可以将它复制到MyXamlPad.exe应用程序中并观察其行为。

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="200" Width="600" WindowStartupLocation="CenterScreen" Title="Growing Label Font!">
  <StackPanel>
    <Label Content = "Interesting...">
      <Label.Triggers>
        <EventTrigger RoutedEvent = "Label.Loaded">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard TargetProperty = "FontSize">
                <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                                 RepeatBehavior = "Forever"/>
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger.Actions>
        </EventTrigger>
      </Label.Triggers>
    </Label>
  </StackPanel>
</Window>
```

下面我们来详细分析这个示例。

30.4.1 演示图板的作用

我们从最里面的元素开始向外分析。首先是<DoubleAnimation>元素，它设置的属性与程序代码中相同（From、To、Duration和RepeatBehavior）：

```
<DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                 RepeatBehavior = "Forever"/>
```

如前所述，Animation元素放置在<Storyboard>元素中，它通过TargetProperty属性将动画对象映射到父类型的某个属性——本例为FontSize。<Storyboard>总是包含在父元素<BeginStoryboard>中。

```
<BeginStoryboard>
  <Storyboard TargetProperty = "FontSize">
    <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                     RepeatBehavior = "Forever"/>
  </Storyboard>
</BeginStoryboard>
```

30.4.2　事件触发器的作用

定义了<BeginStoryboard>元素之后，我们就需要指定一些行为来开始执行动画。WPF包含几种不同的方法在标记中响应运行时条件，其中一种称为触发器。触发器可以理解为在一个较高层次上响应XAML中事件条件的方式，而不需要编写程序代码。

通常，在C#中响应事件时，需要编写事件发生时执行的代码。但触发器则是某个事件发生后的一种通知方式（“我已经加载到内存了”、“鼠标悬浮在我的上方”、“焦点在我这”）。

在事件发生并通知你之后，可以开启演示图板。在本例中，我们响应Label加载到内存的事件。由于我们感兴趣的是Label的Loaded事件，因此<EventTrigger>将放置在Label的触发器集合中：

```
<Label Content = "Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent = "Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "FontSize">
            <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                             RepeatBehavior = "Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

让我们再看一个在XAML中定义动画的示例，这次我们使用关键帧动画。

30.4.3　使用不连续的关键帧创建动画

线性插值动画对象只能在起始点和结束点之间移动，与此不同的是，关键帧可以创建一个包含特殊值的集合，动画将在这些特殊的时间点播放。

为了演示不连续的关键帧类型，我们创建一个Button控件，对其内容执行动画，在3秒的时间内逐字符显示“OK!”。下面的标记位于StringAnimation.xaml文件中，将其复制到MyXamlPad.exe程序并观察输出结果：

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="100" Width="300"
  WindowStartupLocation="CenterScreen" Title="Animate String Data!">
```

```
    <StackPanel>
      <Button Name="myButton" Height="40"
              FontSize="16pt" FontFamily="Verdana" Width = "100">
        <Button.Triggers>
          <EventTrigger RoutedEvent="Button.Loaded">
            <BeginStoryboard>
              <Storyboard>
                <StringAnimationUsingKeyFrames RepeatBehavior = "Forever"
                    Storyboard.TargetName="myButton"
                    Storyboard.TargetProperty="Content"
                    Duration="0:0:3">
                  <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
                  <DiscreteStringKeyFrame Value="O" KeyTime="0:0:1" />
                  <DiscreteStringKeyFrame Value="OK" KeyTime="0:0:1.5" />
                  <DiscreteStringKeyFrame Value="OK!" KeyTime="0:0:2" />
                </StringAnimationUsingKeyFrames>
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger>
        </Button.Triggers>
      </Button>
    </StackPanel>
  </Window>
```

首先我们为按钮定义了一个事件触发器，以确保在按钮加载到内存时能够执行演示图板。StringAnimationUsingKeyFrames类通过Strroryboard.TargetName和Storyboard.TragetProperty属性负责改变按钮的内容。

在<StringAnimationUsingKeyFrames>元素的作用域内，我们定义了4个DiscreteStringKeyFrame元素，它们将在2秒之内改变按钮的Content属性（StringAnimationUsingKeyFrames建立的持续时间为3秒，因此在最后的“!”和下一循环的“O”之间会有一个短暂的停顿）。

现在你应该对如何使用C#代码和XAML构建动画有了更多的了解，下面让我们将注意力转移到WPF样式上来，它大量应用于图形、对象资源和动画之中。

源代码 这些XAML文件位于Chapter 30下的XamlAnimations子目录中。

30.5 WPF 样式的作用

在构建WPF应用程序的UI时，多个控件需要显示同样的外观是很常见的。例如，你可能希望所有的按钮都有同样的高度、宽度、背景颜色以及字体大小。尽管你可以将每个按钮的这些属性都设置成相同的值，但如果以后要修改这些值就会变得异常复杂，因为每次改变你都需要重新设置这些对象的各个属性。

幸运的是，WPF可以使用样式方便地限制相关控件的外观。简而言之，WPF样式是一个对象，它包含一个属性/值对集合。从编程角度来说，一个独立的样式是由System.Windows.Style类来表示的。该类包含一个名为Setters的属性，它是一个Setter对象的强类型集合。Setter就是用来定义属性/值对的对象。

除了Setters集合，Style类还定义了一些重要的成员，可以合并触发器、限制样式所针对的类型甚至基于已知的样式创建新的样式（可以理解为“样式的继承”）。要特别注意以下Style类的成员。

- Triggers：触发器对象的集合，可以获取样式中不同的事件条件。
- BasedOn：可以基于一个已知的样式来构建新的样式。
- TargetType：限制样式所针对的类型。

30.5.1 定义并使用样式

几乎在所有情况下，样式对象都将被打包为对象资源。与其他对象资源一样，可以将其打包为窗体级别或应用程序级别，或者一个专门的资源字典（这使Style对象在应用程序中的访问变得十分方便）。我们的目标是定义一个Style对象，使它至少能够用一组属性/值对来填充一个Setters集合。

在Visual Studio中新建一个WPF应用程序WpfStyles，并创建一个用于所有控件基本字体的样式。打开App.xaml文件，定义如下命名的样式：

```
<Application x:Class="WpfStyles.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">

  <Application.Resources>
    <Style x:Key ="BasicControlStyle">
      <Setter Property = "Control.FontSize" Value ="14"/>
      <Setter Property = "Control.Height" Value = "40"/>
      <Setter Property = "Control.Cursor" Value = "Hand"/>
    </Style>
  </Application.Resources>

</Application>
```

注意，我们为BasicControlStyle的内部集合添加了3个Setter对象。现在，我们在主窗体中对一些控件应用该样式。由于该样式为对象资源，要应用它的对象资源需要使用{StaticResources}或{DynamicResources}标记扩展来定位样式。找到样式之后，它们会用这个资源项设置同样的属性Style。考虑如下的<Window>定义：

```
<Window x:Class="WpfStyles.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="A Window with Style!" Height="229"
  Width="525" WindowStartupLocation="CenterScreen">

  <StackPanel>
    <Label x:Name="lblInfo" Content="This style is boring..."
           Style="{StaticResource BasicControlStyle}" Width="150"/>
    <Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"
           Style="{StaticResource BasicControlStyle}" Width="250"/>
  </StackPanel>

</Window>
```

运行该程序，你会发现两个控件都支持同样的光标、高度和字体大小。

30.5.2 重写样式设置

这里的Button和Label都使用了样式定义的限制。当然，如果控件需要应用一个样式并修改其中的

某些设置，也是可以的。例如，Button现在将使用Help光标（而不是样式中定义的Hand光标）：

```
<Button x:Name="btnTestButton" Content="Yes, but we are reusing settings!"
        Cursor="Help" Style="{StaticResource BasicControlStyle}" Width="250" />
```

控件属性的设置是在处理样式之后进行的，因此控件可以逐个“重写”这些设置。

30.5.3 使用TargetType自动应用样式

当前样式的定义方式是任何控件都可以使用（并需要显式设置控件的Style属性），因为所设置的每个属性都属于Control类。一个程序往往会定义很多设置，这就将导致大量的重复代码。要想使这些样式简洁一些，方法之一是使用TargetType特性。将该特性添加到Style的开放元素中，只需要指定一次该样式可以应用的对象：

```
<Style x:Key ="BasicControlStyle" TargetType="Control">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Height" Value = "40"/>
  <Setter Property = "Cursor" Value = "Hand"/>
</Style>
```

说明 在对基类型构建样式时，你不必关心派生类型是否支持某个依赖属性。如果不支持，将会自动忽略。

这有些帮助，但样式还是可以用于任何控件。在你需要定义用于特殊控件类型的样式时，TargetType特性尤其有用。在应用程序的资源字典中添加如下的样式：

```
<Style x:Key ="BigGreenButton" TargetType="Button">
  <Setter Property = "FontSize" Value ="20"/>
  <Setter Property = "Height" Value = "100"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Background" Value = "DarkGreen"/>
  <Setter Property = "Foreground" Value = "Yellow"/>
</Style>
```

该样式将仅对Button控件生效（或Button的子类），如果将其应用到不兼容的元素中，将得到标记和编译器错误。如果Button像下面这样使用新样式：

```
<Button x:Name="btnTestButton" Content="OK!"
        Cursor="Help" Style="{StaticResource BigGreenButton}" Width="250" />
```

输出结果如图30-12所示。

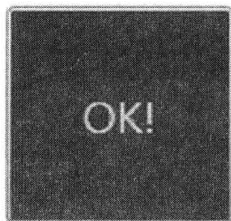


图30-12 使用不同样式的控件

30

30.5.4　继承已有的样式

你还可以使用BasedOn属性根据已有样式构建新的样式。被扩展的样式在字典中必须有一个合适的x:Key，派生的样式在{StaticResource}标记扩展中通过名称引用它。以下是基于BigGreenButton构建的新样式，它将按钮元素旋转20°：

```
<!-- 该样式基于BigGreenButton -->
<Style x:Key ="TiltButton" TargetType="Button" BasedOn = "{StaticResource BigGreenButton}">
  <Setter Property = "Foreground" Value = "White"/>
  <Setter Property = "RenderTransform">
    <Setter.Value>
      <RotateTransform Angle = "20"/>
    </Setter.Value>
  </Setter>
</Style>
```

这时，输出结果如图30-13所示。

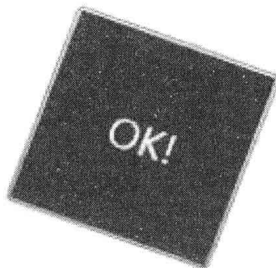


图30-13　使用派生的样式

30.5.5　未命名样式的作用

假设你希望所有的TextBox控件都有同样的外观，那么可以定义一个样式作为应用程序级别的资源，这样程序中的所有窗口都能访问该资源。尽管这么做可行，但如果你有成百上千个窗口包含成百上千个TextBox控件，你就需要对Style属性设置成百上千次！

在一个给定的XAML作用域内，WPF样式可以隐式地应用于所有控件。要创建这样一个样式，只需要使用TargetType属性而不用设置Style资源的x:Key值。这种“未命名的样式”将应用于所有类型正确的控件。以下是另一个应用程序级别的样式，它将自动应用于当前应用程序中所有的TextBox控件。

```
<!-- 所有文本框的默认样式 -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
</Style>
```

现在我们可以定义任意多的TextBox控件，它们将自动显示这个定义好的外观。如果某个TextBox

不想使用该默认外观，可以将Style属性设置为{x:Null}。例如，txtTest将使用默认的未命名样式，而txtTest2将显示自己的外观：

```
<TextBox x:Name="txtTest"/>
<TextBox x:Name="txtTest2" Style="{x:Null}" BorderBrush="Black"
                BorderThickness="5" Height="60" Width="100" Text="Ha!"/>
```

30.5.6 使用触发器定义样式

WPF样式也可以包含触发器，Trigger对象可以被包裹在Style对象的Triggers集合中。在样式中使用触发器可以定义一些<Setter>元素，只有在给定触发器的条件为true时，这些样式才会生效。例如，你可能需要在鼠标悬停在按钮上时放大字体的颜色，或者对当前获取焦点的文本框用某种颜色进行高亮显示。这时触发器十分有用，你可以在某个属性改变时执行特殊的行为，而不必在后台代码文件中显式编写C#代码。

以下是修改后的TextBox样式，当它获得输入焦点时背景色将变为黄色：

```
<!-- 所有文本框的默认样式 -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
  <!-- 下面的setter将只在文本框获得焦点时生效-->
  <Style.Triggers>
    <Trigger Property = "IsFocused" Value = "True">
      <Setter Property = "Background" Value = "Yellow"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

测试该样式，你会发现在不同的TextBox对象间切换时，当前选中的TextBox将显示明亮的黄色背景（只要没有将Style属性设置为{x:Null}）。

属性触发器还很智能，当触发器条件不为true时，属性将自动接受默认的设置。因此，只要TextBox没有获取焦点，它将自动显示默认的颜色。相反，事件触发器（在介绍WPF动画时进行过讨论）不会自动恢复到以前的状态。

30

30.5.7 使用多个触发器定义样式

你还可以这样设计触发器，即当多个条件为true时（类似构建包含多个条件的if语句），我们定义的<Setter>元素才会被应用。例如，我们需要在TextBox获得焦点并且鼠标悬浮在其边框之内时，将它的背景色设置为黄色。我们使用<MultiTrigger>元素来定义每个条件：

```
<!-- 所有文本框的默认样式 -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Height" Value = "30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
```

```
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>
  <!-- 以下的setter只有在文本框获取焦点并且鼠标悬停在文本框之上时才会执行-->
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
          <Condition Property = "IsFocused" Value = "True"/>
          <Condition Property = "IsMouseOver" Value = "True"/>
        </MultiTrigger.Conditions>
      <Setter Property = "Background" Value = "Yellow"/>
    </MultiTrigger>
  </Style.Triggers>
</Style>
```

30.5.8　动画样式

样式还可以合并启动动画序列的触发器。例如如下的样式，将其应用于Button控件，当鼠标悬停在按钮表面时，按钮将会改变大小：

```
<!-- 放大的按钮样式-->
<Style x:Key = "GrowingButtonStyle" TargetType="Button">
  <Setter Property = "Height" Value = "40"/>
  <Setter Property = "Width" Value = "100"/>
  <Style.Triggers>
    <Trigger Property = "IsMouseOver" Value = "True">
      <Trigger.EnterActions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "Height">
            <DoubleAnimation From = "40" To = "200"
                             Duration = "0:0:2" AutoReverse="True"/>
          </Storyboard>
        </BeginStoryboard>
      </Trigger.EnterActions>
    </Trigger>
  </Style.Triggers>
</Style>
```

这里，触发器集合的触发条件是IsMouseOver属性返回true。如果条件成立，我们定义的<Trigger.EnterActions>元素会执行简单的演示图板，在2秒之内将按钮的Height值改为200（然后再改回40）。如果你还需要改变其他的属性，可以定义一个<Trigger.ExitActions>作用域，并在内部放置一些当IsMouseOver为false时的自定义行为。

30.5.9　以编程方式设置样式

应用样式也可以在运行时进行。如果要让最终用户选择UI的外观，或需要强制使用安全设置（例如DisableAllButton样式）的外观，这将非常有帮助。

在本项目中，你已经定义了很多样式，它们大多数都可以应用于Button控件。现在，我们来重组主窗体的UI，使用户在ListBox中选择这些样式。我们将基于用户的选择来应用相应的样式。以下是新的（也是最终的）<Window>元素的标记：

```
<Window x:Class="WpfStyles.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    Height="350" Title="A Window with Style!"
    Width="525" WindowStartupLocation="CenterScreen">

    <DockPanel >
      <StackPanel Orientation="Horizontal" DockPanel.Dock="Top">
        <Label Content="Please Pick a Style for this Button" Height="50"/>
        <ListBox x:Name ="lstStyles" Height ="80" Width ="150" Background="LightBlue"
                 SelectionChanged ="comboStyles_Changed" />
      </StackPanel>
      <Button x:Name="btnStyle" Height="40" Width="100" Content="OK!"/>
    </DockPanel>

</Window>
```

ListBox控件（名称为lstStyles）将在窗体的构造函数中动态填充：

```
public MainWindow()
{
  InitializeComponent();

  // 用所有的Button样式填充下拉列表
  lstStyles.Items.Add("GrowingButtonStyle");
  lstStyles.Items.Add("TiltButton");
  lstStyles.Items.Add("BigGreenButton");
  lstStyles.Items.Add("BasicControlStyle");
}
```

最后我们在相应的代码文件中处理SelectionChanged事件。代码如下，注意我们是如何使用继承的TryFindResource()方法通过名称提取当前资源的：

```
private void comboStyles_Changed(object sender, SelectionChangedEventArgs e)
{
  // 在下拉列表中获取选中的样式名称
  Style currStyle = (Style)
    TryFindResource(lstStyles.SelectedValue);
  if (currStyle != null)
  {
    // 设置按钮类型的样式
    this.btnStyle.Style = currStyle;
  }
}
```

启动该应用程序，你可以在运行时选择这4种按钮样式之一。图30-14显示了完整的应用程序。

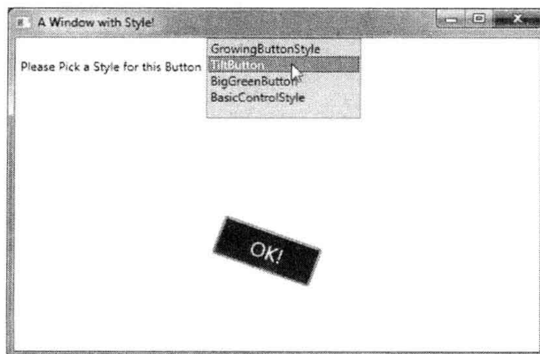


图30-14　使用不同样式的控件

源代码　WpfStyles项目的源代码位于Chapter 30子目录下。

30.6　小结

本章第一部分介绍了WPF的资源管理系统。我们首先介绍了如何使用二进制资源，紧接着介绍了对象资源的作用。正如你所学习到的那样，对象资源是XAML的二进制对象，可以存储在不同的位置，以便复用。

接下来我们学习了WPF的动画框架。我们分别使用C#代码和XAML创建了一些动画。如果使用标记定义动画，需要使用`<storyboard>`元素和触发器来控制动画的执行。最后我们介绍了在图形、对象资源和动画中大量使用的WPF样式机制。

第31章

依赖属性、路由事件和模板

31

本章是介绍WPF编程模型的最后一章，将介绍构建自定义控件的过程。尽管内容模型和样式机制可以为标准的WPF控件添加一些独特的约束，但在构建自定义模板和UserControl的过程中，我们可以完全定义控件呈现输出结果、响应状态转换、以及融入WPF API的方式。

本章开始介绍了创建自定义控件时两个比较重要的话题，依赖属性（dependency property）和路由事件（routed event）。理解了这两个话题之后，将学习默认模板的作用以及如何在运行时以编程方式查看它们。奠定了这个基础之后，本章的剩余部分将介绍如何构建自定义控件，以及如何使用WPF触发器框架添加视觉线索。

说明　构建产品级模板肯定需要使用Expression Blend，因为它内嵌了大量模板工具。在这里我们构建一些简单的可以用Visual Studio创建的模板。再次说明，关于Blend IDE的详细内容，请参考拙作*Pro Expression Blend*（Apress 2011）。

31.1　依赖属性的作用

与其他.NET API一样，WPF在内部实现中使用了.NET类型系统中的所有类型（类、结构、接口、委托、枚举）和所有类型成员（属性、方法、事件、常量、只读字段等）。不过，WPF还提供了一个其他API里没有的编程概念——依赖属性。

与“普通”的.NET属性（通常在WPF的文献中称为CLR属性）一样，依赖属性可以通过XAML或C#代码进行设置，它们最终都封装了类的数据字段，并且可以设置为只读、只写或可读写。

更有趣的是，在大多数情况下你都不会意识到所设置（或访问）的其实是依赖属性，而不是CLR属性。例如，WPF控件从FrameworkElement继承的Height和Width属性，以及从ControlContent继承的Content成员，实际上都是依赖属性：

```
<!-- 设置依赖属性 -->
<Button x:Name = "btnMyButton" Height = "50" Width = "100" Content = "OK"/>
```

既然如此相似，WPF为什么还要定义这样一个新的概念呢？答案在于依赖属性在类的内部是如何实现的。稍后你将看到代码示例，而此时，我们从一个更高的层次来看看依赖属性所遵循的创建方式。

- 首先，定义依赖属性的类在其继承链中必须包含DependencyObject。
- 一个依赖属性在类中表示为一个公共的、静态的、类型为DependencyProject的只读字段。按

照惯例，该字段的命名方式为CLR属性名加上Property后缀（参见最后一条）。

- 通过调用静态方法DependencyProperty.Register()可以注册DependencyProperty变量，这通常发生在静态构造函数中或内联于变量声明时。
- 最后，将在类中定义一个XAML友好的CLR属性，它调用DependencyObject提供的方法来获取或设置值。

依赖属性的实现提供了众多强大的特性，可以应用于各种WPF技术之中，包括数据绑定、动画服务、样式等。简而言之，依赖属性的目的是提供一种方式，用来计算基于其他输入值的属性的值。以下是依赖属性的主要优点，它们的功能远不止是CLR属性对于数据的简单封装。

- 依赖属性可以从XAML定义的父元素中继承值。例如，如果在开放的<Window>标记中定义了FontSize特性的值，那么该Window中所有的控件默认字体大小都是相同的。
- 依赖属性允许由XAML作用域内的元素来设置值，例如Button可以设置父元素DockPanel的Dock属性。（类似第28章中的*附加属性*，其实附加属性也是依赖属性的一种形式。）
- 依赖属性允许WPF通过多个外部的值来计算属性值，这对于动画和数据绑定服务是非常重要的。
- 依赖属性提供了WPF触发器的底层支持（同样也常用于动画和数据绑定）。

记住，在很多情况下与依赖属性的交互和与普通CLR属性的交互是完全一样的（这有赖于XAML的封装）。但是，在介绍第28章中的数据绑定时，我们了解到如果要在代码中建立数据绑定，必须调用目标对象的SetBinding()方法，并指定要操作的*依赖属性*：

```
private void SetBindings()
{
  Binding b = new Binding();
  b.Converter = new MyDoubleConverter();
  b.Source = this.mySB;
  b.Path = new PropertyPath("Value");

  // 指定依赖属性
  this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

在代码中启动动画的代码也与之类似，例如第30章的：

```
// 指定依赖属性
rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
```

只有在编写自定义WPF控件时，你才会需要建立自定义依赖属性。例如，如果你正在构建一个包含4个自定义属性的UserControl，并需要这些属性能够和WPF API很好地集成，这时就需要用依赖属性逻辑来进行编码。

具体来说，如果自定义属性需要使用数据绑定或操作动画，或者在发生改变时需要广播，或者必须像WPF样式中的Setter那样工作，或者必须能够从父元素那里接收值，那么普通的CLR属性将不能满足要求。如果某个程序员使用了普通的CLR属性，那么其他程序员能够获取或设置普通CLR属性的值；但如果其他程序员要在WPF服务的上下文中使用这些属性，将不会像期望的那样。因为那个程序员永远无法了解其他人希望如何与自定义UserControl类中的属性进行交互，因此在构建自定义控件时最好养成总是定义依赖属性的习惯。

31.1.1 已知的依赖属性

在学习如何构建自定义依赖属性之前，我们先来看看FrameworkElement类中的Height属性在内部是如何实现的。相关代码如下所示（包括注释）。

```
// FrameworkElement是一个 (is-a) DependencyObject
public class FrameworkElement : UIElement, IFrameworkInputElement,
  IInputElement, ISupportInitialize, IHaveResources, IQueryAmbient
{
...
  // DependencyProperty类型的静态只读字段
  public static readonly DependencyProperty HeightProperty;

  // 字段常常在类的静态构造函数中进行注册
  static FrameworkElement()
  {
    ...
    HeightProperty = DependencyProperty.Register(
      "Height",
      typeof(double),
      typeof(FrameworkElement),
      new FrameworkPropertyMetadata((double) 1.0 / (double) 0.0,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
      new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
  }

  // CLR包装器，通过继承的GetValue()/SetValue()方法实现
  public double Height
  {
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
  }
}
```

如你所见，依赖属性比普通CLR属性所需的代码要多得多！而实际上也可能比这更复杂（幸运的是，许多实现都要比Height简单）。

首先，如果一个类要定义依赖属性，那么它的继承链上必须有DependencyObject，该类定义了CLR包装器所需的GetValue()和SetValue()方法。FrameworkElement是一个（is-a）DependencyObject，可以满足该要求。

其次，保存实际属性值（本例中的Height为double类型）的实体表示为一个公共的、静态只读的DependencyProperty类型的字段。按照惯例，该字段的名称通常为相关的CLR包装器名称加上Property作为后缀，例如：

```
public static readonly DependencyProperty HeightProperty;
```

由于依赖属性为静态字段，它们通常在类的静态构造函数内部创建（和注册）。DependencyProperty对象由静态方法DependencyProperty.Register()创建。该方法包含多个重载。不过对于Height来说，DependencyProperty.Register()的调用如下：

```
HeightProperty = DependencyProperty.Register(
  "Height",
  typeof(double),
  typeof(FrameworkElement),
```

31

```
  new FrameworkPropertyMetadata((double)0.0,
    FrameworkPropertyMetadataOptions.AffectsMeasure,
    new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
  new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
```

DependencyProperty.Register()方法的第一个参数为该类中普通CLR属性的名称（本例中为Height）。第二个参数为它封装的数据类型的类型信息（double）。第三个参数指定了该属性所属的类的类型信息（FrameworkElement）。这看起来好像有些多余（毕竟FrameworkElement类已经定义了HeightProperty字段），但这正是WPF的绝妙之处，你可以在一个类中注册另一个类的依赖属性（即使那个类是封闭的！）。

DependencyProperty.Register()的第四个参数真正赋予了依赖属性独有的特色。FrameworkPropertyMetadata对象描述了各种细节，包括WPF如何处理该属性的回调通知（如果属性在更改时需要进行通知），以及由FrameworkPropertyMetadataOptions枚举表示的选项，用来控制属性的行为（是否可以进行数据绑定，可否被继承，等等）。FrameworkPropertyMetadata可以像下面这样分解：

```
new FrameworkPropertyMetadata(
  // 属性的默认值
  (double)0.0,

  // 元数据选项
  FrameworkPropertyMetadataOptions.AffectsMeasure,

  // 属性更改时调用的委托
  new PropertyChangedCallback(FrameworkElement.OnTransformDirty)
)
```

FrameworkPropertyMetadata构造函数的最后一个参数为一个委托，它的构造函数的参数指向了FrameworkElement类中的静态方法OnTransformDirty()。我不想列出该方法的后台代码，但要记住的是任何时候我们要创建自定义依赖属性，都可以指定PropertyChangedCallback委托，使其指向一个方法，该方法会在属性值发生改变的时候调用。

DependencyProperty.Register()方法的最后一个参数是一个ValidateValueCallback类型的委托，它指向了FrameworkElement类中的一个方法，用来确保为属性设置的值是有效的：

```
new ValidateValueCallback(FrameworkElement.IsWidthHeightValid)
```

我们习惯将该方法内的逻辑放置在属性的set块中（详细内容将在下节介绍）：

```
private static bool IsWidthHeightValid(object value)
{
  double num = (double) value;
  return ((!DoubleUtil.IsNaN(num) && (num >= 0.0))
    && !double.IsPositiveInfinity(num));
}
```

注册了DependencyProperty对象之后，剩下的任务就是用普通的CLR属性（本例中为Height）封装该字段。但要注意的是，get和set作用域并不是简单地返回或设置类级别的double成员变量，而是间接地使用System.Windows.DependencyObject基类的GetValue()和SetValue()方法：

```
public double Height
{
  get { return (double) base.GetValue(HeightProperty); }
  set { base.SetValue(HeightProperty, value); }
}
```

31.1.2 CLR属性包装器的重要说明

在XAML或代码中获取或设置依赖属性与操作普通的属性没什么区别，但是它们的后台实现却使用了更加复杂的编码技术。记住，我们之所以这么做，是想构建一个包含自定义属性的自定义控件，该控件通过这些自定义属性与那些需要和依赖属性通信的WPF服务（如动画、数据绑定、样式）进行集成。

尽管定义CLR包装器是依赖属性实现的一部分，但你不能在set块中添加验证逻辑。并且，依赖属性的CLR包装器除了调用GetValue()和SetValue()之外也不能做其他任何事情。

这是由WPF运行时的构造方式决定的。当你用XAML设置属性时，如

```
<Button x:Name="myButton" Height="100" .../>
```

运行时将完全绕过Height属性的set块而直接调用SetValue()方法！这么做尽管怪异，但却可以进行简单的优化。如果WPF运行时直接访问Height属性的set块，需要执行运行时反射才能找到DependencyProperty字段（由SetValue()的第一个参数指定）的位置，然后在内存中引用，等等。同样，在XAML中获取Height属性的值，也是直接调用了GetValue()方法。

既然这样，为什么还要构建CLR包装器呢？WPF XAML不允许在标记中调用函数，因此下面的标记会报错：

```
<!-- 不！不能在WPF XAML中调用方法 -->
<Button x:Name="myButton" this.SetValue("100") .../>
```

当你在标记中使用CLR包装器设置或获取值时，实际上是在告诉WPF运行时："嘿！去调用GetValue()/SetValue()，在标记中不能直接访问我！"那么，在代码中调用CLR包装器的情况如何呢：

```
Button b = new Button();
b.Height = 10;
```

在这种情况下，不再涉及WPF XAML优化，因此Height属性的set块中调用SetValue()方法之外的其他代码也会被执行。

需要记住的一条基本规则是，由于在注册依赖属性时，使用ValidateValueCallback委托指定了一个执行数据校验的方法，因此可以确保行为的正确性，而不必关心是使用XAML还是代码来获取或设置依赖属性。

31.2 构建自定义依赖属性

如果你对本章以上内容感到有点头疼，这完全是正常的反应。你需要花点时间才能适应依赖属性。但是无论如何，这是构建许多自定义WPF控件所必须了解的部分，因此我们来看看如何创建依赖属性。

新建一个WPF应用程序CustomDepPropApp，打开Project菜单，单击Add User Control菜单选项，创建一个名为ShowNumberControl.xaml的控件（如图31-1所示）。

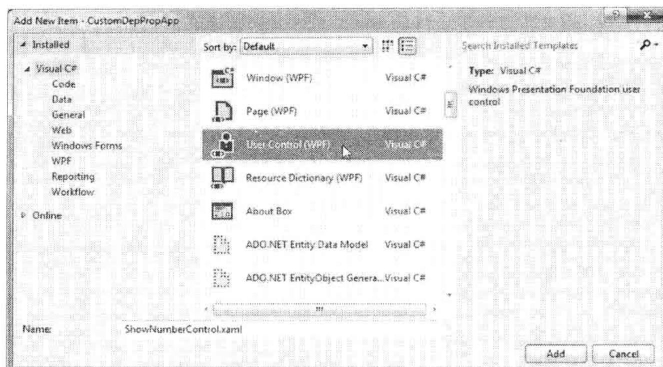


图31-1　添加新的自定义UserControl

说明　本章后面将详细介绍WPF UserContrl，现在只需要按部就班地跟着做就可以了。

和窗体一样，WPF UserControl类型也包含一个XAML文件和一个相关的代码文件。更新控件的XAML，在Grid内定义一个Label控件：

```
<UserControl x:Class="CustomDepPropApp.ShowNumberControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="300">
  <Grid>
    <Label x:Name="numberDisplay" Height="50" Width="200" Background="LightBlue"/>
  </Grid>
</UserControl>
```

在该自定义控件的代码文件中创建一个普通的.NET属性，该属性的类型为int，并且使用新的值设置Label的Content属性：

```
public partial class ShowNumberControl : UserControl
{
  public ShowNumberControl()
  {
    InitializeComponent();
  }

  // 普通的.NET属性
  private int currNumber = 0;
  public int CurrentNumber
  {
    get { return currNumber; }
    set
    {
      currNumber = value;
      numberDisplay.Content = CurrentNumber.ToString();
    }
  }
}
```

现在，更新窗体的XAML定义，在StackPanel布局管理器内声明自定义控件的实例。由于我们的自定义控件不属于核心WPF程序集栈，你需要自定义一个XAML命名空间来映射控件（详见第27章）。标记如下所示：

```
<Window x:Class="CustomDepPropApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:myCtrls="clr-namespace:CustomDepPropApp"
  Title="Simple Dependency Property App" Height="150" Width="250"
  WindowStartupLocation="CenterScreen">

  <StackPanel>
    <myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100"/>
  </StackPanel>
</Window>
```

如你所见，Visual Studio设计器可以正确显示我们设置的CurrentNumber属性的值（如图31-2所示）。

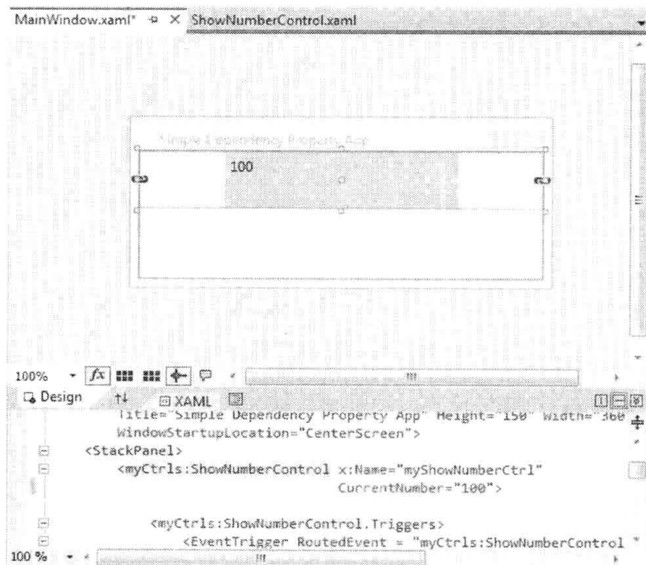


图31-2 属性显示正常

但是，如果对CurrentNumber属性执行一个动画，使其值在10秒内从100变为200会如何呢？想要在标记中实现，就要像下面这样修改<myCtrls:ShowNumberControl>作用域：

```
<myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100">
  <myCtrls:ShowNumberControl.Triggers>
    <EventTrigger RoutedEvent = "myCtrls:ShowNumberControl.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "CurrentNumber">
            <Int32Animation From = "100" To = "200" Duration = "0:0:10"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
```

31

```
    </EventTrigger>
  </myCtrls:ShowNumberControl.Triggers>
</myCtrls:ShowNumberControl>
```

运行应用程序，动画对象无法找到合适的目标，因此将被忽略。这是因为CurrentNumber属性没有被注册为依赖属性！要解决这个问题，回到自定义控件的代码文件中，将当前属性的逻辑全部注释掉（包括私有字段）。现在，将鼠标光标放到类的作用域内，输入propdp代码段（如图31-3所示）。

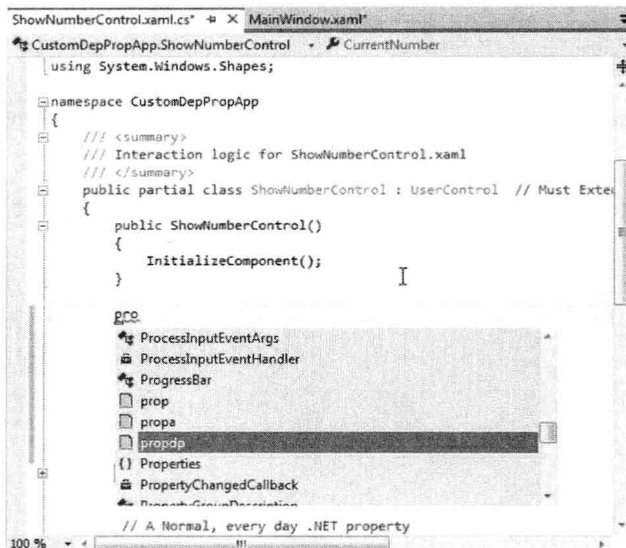


图31-3　propdp代码段是创建依赖属性的第一步

输入propdp后，连按两次Tab键。代码段展开成依赖属性的基本架构（如图31-4所示）。

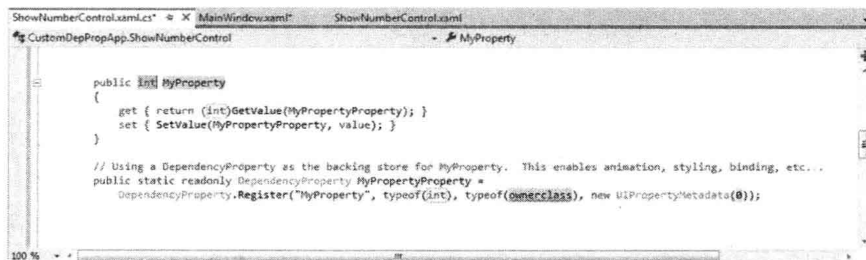


图31-4　展开的代码段

最简单的CurrentNumber属性如下所示：

```
public partial class ShowNumberControl : UserControl
{
  public int CurrentNumber
  {
    get { return (int)GetValue(CurrentNumberProperty); }
    set { SetValue(CurrentNumberProperty, value); }
  }
```

```
    public static readonly DependencyProperty CurrentNumberProperty =
      DependencyProperty.Register("CurrentNumber",
      typeof(int),
      typeof(ShowNumberControl),
      new UIPropertyMetadata(0));
   ...
}
```

这与Height属性的实现十分类似，只不过代码段用内联的方式注册属性，而不是在静态构造函数中（两种方法皆可）。同时还要注意这里使用UIPropertyMetadata对象定义整型的默认值（0），而不要使用复杂的FrameworkPropertyMetadata对象。

31.2.1 添加数据验证例程

尽管现在已经有了依赖属性CurrentNumber，但动画还是无法执行。接下来要调整的是指定一个函数来执行一些数据验证逻辑。例如，确保CurrentNumber的值介于0~500之间。

我们为DependencyProperty.Register()方法添加最后一个参数ValidateValueCallback，将其指向ValidateCurrentNumber方法。

ValidateValueCallback是一个委托，它指向包含唯一的object类型参数并返回bool的方法。这里的object代表设置的新值。ValidateCurrentNumber()方法判断该值是否在允许的范围内，并返回true或false：

```
public static readonly DependencyProperty CurrentNumberProperty =
  DependencyProperty.Register("CurrentNumber", typeof(int), typeof(ShowNumberControl),
  new UIPropertyMetadata(100),
  new ValidateValueCallback(ValidateCurrentNumber));

public static bool ValidateCurrentNumber(object value)
{
  // 简单的业务逻辑。value必须介于0~500之间
  if (Convert.ToInt32(value) >= 0 && Convert.ToInt32(value) <= 500)
    return true;
  else
    return false;
}
```

31.2.2 响应属性的改变

现在我们验证了数字的有效性，但还是无法显示动画。我们需要指定UIPropertyMetadata构造函数的第二个参数PropertyChangedCallback对象。该委托所指向的方法的第一个参数为DependencyObject类型，第二个参数为DependencyPropertyChangedEventArgs类型。首先，将代码改为如下形式：

```
// 注意UIPropertyMetadata构造函数的第二个参数
public static readonly DependencyProperty CurrentNumberProperty =
  DependencyProperty.Register("CurrentNumber", typeof(int), typeof(ShowNumberControl),
  new UIPropertyMetadata(100,
                         new PropertyChangedCallback(CurrentNumberChanged)),
  new ValidateValueCallback(ValidateCurrentNumber));
```

在CurrentNumberChanged()方法内，我们的最终目标是按照CurrentNumber属性设定的值修改Label

的Content。但一个很大的问题是，CurrentNumberChanged()方法必须是静态的，因为它被静态的DependencyProperty对象所使用。那么如何访问当前ShowNumberControl的实例呢？答案是使用第一个DependencyObject类型的参数，而新的值可以从接下来的事件参数中得到。以下是改变Label的Content属性所必需的代码：

```
private static void CurrentNumberChanged(DependencyObject depObj,
    DependencyPropertyChangedEventArgs args)
{
  // 将DependencyObject转换为ShowNumberControl
  ShowNumberControl c = (ShowNumberControl)depObj;

  // 从ShowNumberControl中获取Label控件
  Label theLabel = c.numberDisplay;

  // 用新的值设置Label
  theLabel.Content = args.NewValue.ToString();
}
```

哎呀！仅仅改变Label的输出就要这么麻烦。但这么做的好处是依赖属性CurrentNumber现在可以应用WPF样式、动画和数据绑定操作，等等。如果运行应用程序，你应该能看见值不停地改变。）

这就是关于WPF依赖属性的内容。我希望你对这些结构的功能以及如何创建它们能有更好的了解，但其实还有很多细节没有涵盖在内。

如果你要构建大量包含自定义属性的自定义控件，请查阅.NET Framework 4.5 SDK文档中WPF Fundamentals部分的Properties话题。该文档包含大量示例，如构建依赖属性、附加属性，配置属性元数据的不同方式，等等。

源代码 CustomDepPropApp项目的源代码位于Chapter 31子目录下。

31.3 路由事件

属性并不是.NET编程构造为了适应WPF API所进行的唯一调整。标准的CLR事件模型也做了进一步的完善，以确保事件的处理方式能够适应对象树的XAML描述。假设有一个新的WPF应用程序项目WPFRoutedEvents。更新初始窗体的XAML描述，添加如下所示的定义了一些复杂内容的<Button>控件：

```
<Button Name="btnClickMe" Height="75" Width = "250"
        Click ="btnClickMe_Clicked">
  <StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100" >
    <Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
             Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
    <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
             Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

注意，在<Button>的开放式定义中我们处理了Click事件，指定了在事件触发时调用的方法名称。Click事件为RoutedEventHandler委托服务，它要求事件处理程序的第一个参数为object，第二个参数为System.Windows.RoutedEventArgs。处理程序的实现如下所示：

```
public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
{
  // 单击按钮时的操作
  MessageBox.Show("Clicked the button");
}
```

运行应用程序，单击按钮内容的任何部位（绿色的Ellipse、黄色的Ellipse、Label或Button的表面）都会显示该消息框。这很好。想象一下如果你必须为每个子元素添加Click事件处理程序，将是多么枯燥的事情。为Button的各个部分创建各自的事件处理程序不仅将带来繁重的工作量，而且最终的代码也难以维护。

幸运的是，WPF路由事件（routed event）将确保单击按钮的任何部分都能自动调用唯一的Click事件处理程序。简而言之，路由事件模型将在对象树中自动向上（或向下）传递事件，来寻找适当的处理程序。

具体来说，一个路由事件可以使用三种路由策略（routing strategy）。如果事件在对象树中从初始的位置向上移动到其他作用域，这种事件称为冒泡事件（bubbling event）。相反，如果事件从最外层元素（如Window）向下移动到初始位置，这种事件称为隧道事件（tunneling event）。最后，如果事件仅仅在初始元素中触发并处理（可认为是普通的CLR事件），这种事件称为直接事件（direct event）。

31.3.1 路由冒泡事件的作用

在本例中，如果用户单击黄色椭圆，Click事件冒泡到上一层作用域（Canvas），然后是StackPanel，最后到达Click事件处理程序所在的Button。同样，如果用户单击Label，事件冒泡到StackPanel，最后到达Button元素。

有了这种冒泡路由事件模式，就不需要为组合控件的所有成员分别注册具体的Click事件处理程序。不过，如果你希望对同一棵对象树上的多个元素执行不同的单击逻辑，也是可以实现的。

假设你需要对outerEllipse控件执行不同的单击逻辑。首先处理该子元素的MouseDown事件（像Ellipse这样的图形呈现类型不支持单击事件，但它们可以通过MouseDown、MouseUp等事件监控鼠标按钮的活动）：

```
<Button Name="btnClickMe" Height="75" Width = "250"
        Click ="btnClickMe_Clicked">
  <StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100" >
    <Ellipse Name = "outerEllipse" Fill ="Green"
             Height ="25" MouseDown ="outerEllipse_MouseDown"
             Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
    <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
             Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

然后实现一个适当的事件处理程序，为简便起见，我们只改变主窗体的Title属性：

31

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
  // 修改窗体的标题
  this.Title = "You clicked the outer ellipse!";
}
```

这样，你就可以根据用户单击的位置（外部椭圆或按钮的其他范围）而执行不同的行为。

说明　冒泡路由事件总是从起始点移动到下一个定义的作用域。因此在本例中，单击innerEllipse对象，事件将冒泡到Canvas，而不是outerEllipse，因为他们是同在Canvas作用域内的Ellipse类型。

31.3.2　继续或中止冒泡

现在，如果用户单击outerEllipse对象，将触发该Ellipse对象的MouseDown事件处理程序，而此时事件也会冒泡到按钮的Click事件。如果希望通知WPF停止在对象树中的冒泡，可以将EventArgs参数的Handled属性设置为true：

```
public void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
  // 修改窗体标题
  this.Title = "You clicked the outer ellipse!";

  // 终止冒泡
  e.Handled = true;
}
```

这时，你会发现窗体的标题修改了，但Button的Click事件处理程序显示的MessageBox没有弹出。概括地说，路由冒泡事件可以将一组复杂的内容视为一个单独的逻辑元素（如Button）或不相关的项（如Button中的Ellipse）。

31.3.3　路由隧道事件的作用

严格地说，路由事件在本质上既可以冒泡（如前所述）也可以隧道传递。隧道事件（全部以Preview前缀开头，如PreviewMouseDown）从对象树的最外层深入到内层作用域。一般来说，WPF基础类库中每个冒泡事件都有一个与之对应并且在其之前触发的隧道事件。例如，在MouseDown冒泡事件触发前，会先触发PreviewMouseDown隧道事件。

处理隧道事件看上去与处理其他事件没什么不同。只需要在XAML中设置事件处理程序的名称（如果需要的话也可以在代码文件中使用相应的C#事件处理语法），然后在代码文件中实现处理程序。为了演示隧道事件和冒泡事件的相互影响，我们为outerEllipse对象添加PreviewMouseDown事件：

```
<Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
         MouseDown ="outerEllipse_MouseDown"
         PreviewMouseDown ="outerEllipse_PreviewMouseDown"
         Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
```

接下来，在C#类中添加这两个事件处理程序（为所有对象），使用传入的事件参数对象向一个string成员变量mouseActivity中追加数据。这有助于我们观察后台事件的触发顺序：

```
public partial class MainWindow : Window
{
  string mouseActivity = string.Empty;
  public MainWindow()
  {
    InitializeComponent();
  }

  public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
  {
    AddEventInfo(sender, e);
    MessageBox.Show(mouseActivity, "Your Event Info");

    // 为下一轮次清空字符串
    mouseActivity = "";
  }

  private void AddEventInfo(object sender, RoutedEventArgs e)
  {
    mouseActivity += string.Format(
      "{0} sent a {1} event named {2}.\n", sender,
      e.RoutedEvent.RoutingStrategy,
      e.RoutedEvent.Name);
  }

  private void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
  {
    AddEventInfo(sender, e);
  }

  private void outerEllipse_PreviewMouseDown(object sender, MouseButtonEventArgs e)
  {
    AddEventInfo(sender, e);
  }
}
```

注意，这里我们没有在任何一个事件处理程序中停止事件的冒泡过程。运行该程序，单击按钮上不同的位置，将显示不同的消息框。图31-5显示了单击outerEllipse对象时的输出结果：

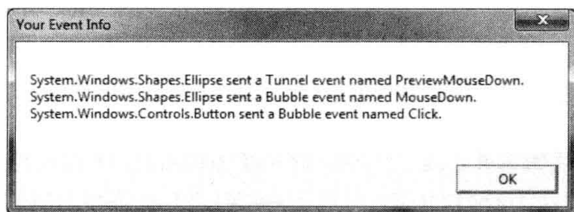


图31-5　先隧道，再冒泡

那么为什么WPF事件要成对出现呢（一个隧道事件和一个冒泡事件）？答案是通过前置事件，你可以在相应的冒泡事件触发之前执行任何特殊的逻辑（数据验证、禁止冒泡行为等）。假设你有一个只允许输入数字的TextBox，你可以处理PreviewKeyDown事件，如果发现输入的数据不是数字，就可以将Handled属性设置为true来取消冒泡事件。

正如你所想的那样，当构建包含自定义事件的自定义控件时，你可以编写类似的可以在XAML树

中冒泡（或隧道传递）的事件。本章将不介绍如何构建自定义路由事件（其过程与构建自定义依赖属性没什么太大区别）。如果感兴趣，可以浏览.NET Framework 4.5 SDK文档中关于Routed Events Overview的主题，你可以找到大量辅助教程。

源代码　WPFRoutedEvents项目的源代码位于Chapter 31子目录下。

31.4　逻辑树、可视树和默认模板

在开始学习如何构建自定义控件之前，还有一些需要研究的话题。特别是需要了解逻辑树（logical tree）、可视树（visual tree）和默认模板（default template）之间的区别。当你在Visual Studio或类似kaxaml.exe这样的工具中编写XAML时，标记就是XAML文档的逻辑视图。同样，当你用C#代码向StackPanel控件中添加新项时，实际上是在向逻辑树中添加新项。从本质上讲，逻辑视图表示主Window（或其他根元素如Page、NavigationWindow）是如何在不同的布局管理器中定位内容的。

然而，在每棵逻辑树的背后，都有一棵默默工作着的可视树。在后台，WPF用更加复杂的可视树将元素正确呈现到屏幕上。在每个可视树中，都会有更加详尽的模板和样式用来呈现各个对象，包括必要的绘图、形状、可视化和动画。

了解逻辑树和可视树的区别是十分有用的，因为当构建自定义控件模板时，你实际上是在移除某个控件全部或部分默认的可视树，并添加你自己定义的内容。因此，如果需要Button控件呈现为星形形状，你需要定义一个星型模板并将其插入到Button的可视树中。从逻辑角度来说，该Button仍然为Button类型，并且仍旧支持按钮所有的属性、方法和事件。但从视觉角度来说，它却呈现出完全不同的外观。正是该特性使得WPF成为一个非常有用的API，因为其他工具要想制作一个星形按钮都需要构建一个崭新的类，而用WPF则只需要定义一个新的标记。

说明　WPF控件通常被描述为不可见的（lookless）。这是因为WPF控件的外观与其行为是完全独立的（和可定制的）。

31.4.1　以编程方式查看逻辑树

在运行时分析窗体的逻辑树并不是一个常见的WPF编程活动，不过值得一提的是System.Windows命名空间定义了一个LogicalTreeHelper类，可以在运行时查看逻辑树的结构。为了演示逻辑树、可视树和空间模板之间的联系，我们创建一个WPF应用程序TreesAndTemplatesApp。

更新窗体的标记，使其包含两个Button控件和一个很大的只读的、含有滚动条的TextBox。使用IDE为两个按钮添加Click事件处理程序。XAML如下所示：

```
<Window x:Class="TreesAndTemplatesApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Trees and Templates" Height="518"
  Width="836" WindowStartupLocation="CenterScreen">
```

```
  <DockPanel LastChildFill="True">
    <Border Height="50"  DockPanel.Dock="Top" BorderBrush="Blue">
      <StackPanel Orientation="Horizontal">
        <Button x:Name="btnShowLogicalTree" Content="Logical Tree of Window"
                Margin="4" BorderBrush="Blue" Height="40"Click="btnShowLogicalTree_Click"/>
        <Button x:Name="btnShowVisualTree" Content="Visual Tree of Window"
                BorderBrush="Blue" Height="40" Click="btnShowVisualTree_Click"/>
      </StackPanel>
    </Border>
    <TextBox x:Name="txtDisplayArea" Margin="10" Background="AliceBlue" IsReadOnly="True"
             BorderBrush="Red" VerticalScrollBarVisibility="Auto"
             HorizontalScrollBarVisibility="Auto" />
  </DockPanel>

</Window>
```

在C#代码文件中，定义一个string类型的成员变量dataToShow。在btnShowLogicalTree对象的Click处理程序中，调用一个辅助的递归方法，将Window的逻辑树信息赋值给字符串变量。这需要调用LogicalTreeHelper的静态方法GetChildren()。代码如下所示：

```
private void btnShowLogicalTree_Click(object sender, RoutedEventArgs e)
{
  dataToShow = "";
  BuildLogicalTree(0, this);
  this.txtDisplayArea.Text = dataToShow;
}

void BuildLogicalTree(int depth, object obj)
{
  // 将类型的名称追加到dataToShow成员变量中
  dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";

  // 如果该项不是DependencyObject，则跳过
  if (!(obj is DependencyObject))
    return;

  // 为每个逻辑子元素进行递归
  foreach (object child in LogicalTreeHelper.GetChildren(
    obj as DependencyObject))
      BuildLogicalTree(depth + 5, child);
}
```

运行应用程序并单击第一个按钮，你将看到文本框内打印了一个树，这是初始XAML的完美重现（如图31-6所示）。

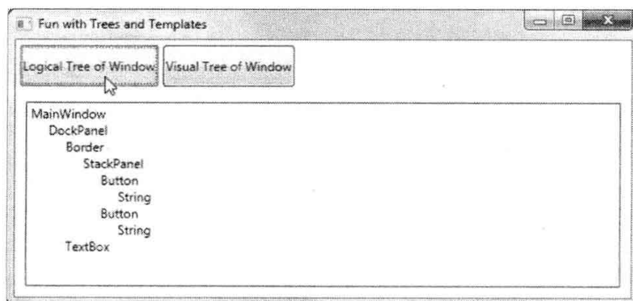


图31-6　在运行时查看逻辑树

31.4.2 以编程方式查看可视树

使用System.Windows.Media命名空间下的VisualTreeHelper类也可以在运行时查看Window的可视树。以下是第二个Button控件（btnShowVisualTree）的Click实现，它执行类似的递归逻辑来构建可视树的文本化描述：

```
private void btnShowVisualTree_Click(object sender, RoutedEventArgs e)
{
  dataToShow = "";
  BuildVisualTree(0, this);
  this.txtDisplayArea.Text = dataToShow;
}

void BuildVisualTree(int depth, DependencyObject obj)
{
  // 将类型的名称追加到dataToShow成员变量中
  dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
  // 为每个可视子节点进行递归
  for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
    BuildVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
}
```

如图31-7，可视树暴露了一些低级别的呈现代理，如ContentPresenter、AdornerDecorator、TextBoxLineDrawingVisual，等等。

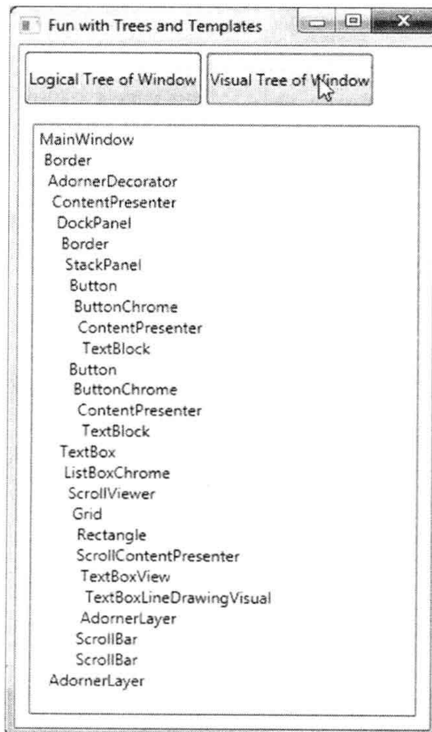


图31-7 在运行时查看可视树

31.4.3 以编程方式查看控件的默认模板

WPF中的可视树可以用来理解如何呈现Window及其包含的元素。每一个WPF控件都在其默认模板内保存了一些呈现指令。以编程角度来说，任何模板都可以表示为一个ControlTemplate类的实例。同样，你也可以通过Template属性获取控件的默认模板：

```
// 获取Button的默认模板
Button myBtn = new Button();
ControlTemplate template = myBtn.Template;
```

同样，你也可以在代码中创建新的ControlTemplate对象，并将其赋值给控件的Template属性：

```
// 为按钮添加一个新的模板
Button myBtn = new Button();
ControlTemplate customTemplate = new ControlTemplate();

// 假设该方法包含创建星型模板的所有代码
MakeStarTemplate(customTemplate);
myBtn.Template = customTemplate;
```

尽管你可以使用代码创建新的模板，但目前比较常见方法的还是在XAML里进行创建。不过，在开始构建你自己的模板之前，我们先来完成当前这个示例，使其能够在运行时查看WPF控件的默认模板。这样可以了解模板的整个组成，是十分有用的。首先，使用新的StackPanel更新窗体标记，其父元素在上一级DockPanel中靠左停靠，代码如下：

```
<Border DockPanel.Dock="Left" Margin="10" BorderBrush="DarkGreen"
        BorderThickness="4" Width="358">
  <StackPanel>
    <Label Content="Enter Full Name of WPF Control" Width="340" FontWeight="DemiBold" />
    <TextBox x:Name="txtFullName" Width="340" BorderBrush="Green"
             Background="BlanchedAlmond" Height="22"
             Text="System.Windows.Controls.Button" />
    <Button x:Name="btnTemplate" Content="See Template" BorderBrush="Green"
            Height="40" Width="100" Margin="5"
            Click="btnTemplate_Click" HorizontalAlignment="Left" />
    <Border BorderBrush="DarkGreen" BorderThickness="2" Height="260"
            Width="301" Margin="10" Background="LightGreen" >
      <StackPanel x:Name="stackTemplatePanel" />
    </Border>
  </StackPanel>
</Border>
```

注意这个空的StackPanel元素stackTemplatePanel，我们将在代码中引用它。现在窗体的外观将如图31-8所示。

左上方的文本区域可以输入位于PresentationFramework.dll程序集中的某个WPF控件的完全限定名称。在库加载之后，你可以动态创建该对象的实例，并在左下方的大块文本区域中显示。最后，控件的默认模板将显示在右侧的文本区域中。首先，在C#类中添加一个Control类型的成员变量：

```
private Control ctrlToExamine = null;
```

以下是剩余的代码，它需要引用System.Reflection、System.Xml和System.Windows.Markup命名空间：

```
private void btnTemplate_Click(object sender, RoutedEventArgs e)
{
  dataToShow = "";
  ShowTemplate();
```

```
    this.txtDisplayArea.Text = dataToShow;
  }

  private void ShowTemplate()
  {
    // 移除在当前预览区域中的控件
    if (ctrlToExamine != null)
      stackTemplatePanel.Children.Remove(ctrlToExamine) ;
    try
    {
      // 加载PresentationFramework, 创建指定控件的实例, 并为它指定一个尺寸用于显示,
      // 然后添加到空的StackPanel中
      Assembly asm = Assembly.Load("PresentationFramework, Version=4.0.0.0," +
        "Culture=neutral, PublicKeyToken=31bf3856ad364e35");
      ctrlToExamine = (Control)asm.CreateInstance(txtFullName.Text);
      ctrlToExamine.Height = 200;
      ctrlToExamine.Width = 200;
      ctrlToExamine.Margin = new Thickness(5);
      stackTemplatePanel.Children.Add(ctrlToExamine);

      // 定义一些XAML设置, 以保持缩进
      XmlWriterSettings xmlSettings = new XmlWriterSettings();
      xmlSettings.Indent = true;

      // 创建一个StringBuilder来保存XAML
      StringBuilder strBuilder = new StringBuilder();

      // 创建一个基于设置的XmlWriter
      XmlWriter xWriter = XmlWriter.Create(strBuilder, xmlSettings);

      // 将基于ControlTemplate的XAML保存到XmlWriter对象
      XamlWriter.Save(ctrlToExamine.Template, xWriter);

      // 在文本框中显示XAML
      dataToShow = strBuilder.ToString();
    }
    catch (Exception ex)
    {
      dataToShow = ex.Message;
    }
  }
```

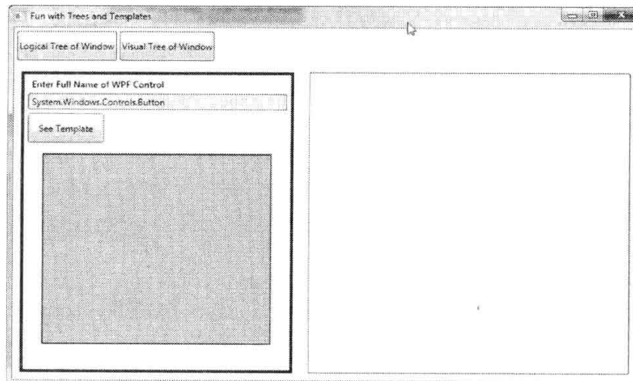


图31-8　更新后的窗体UI

这段代码的大部分工作是将已编译的BAML资源映射到XAML字符串。图31-9演示了应用程序最终的情形，显示了System.Windows.Controls.DatePicker控件的默认模板。

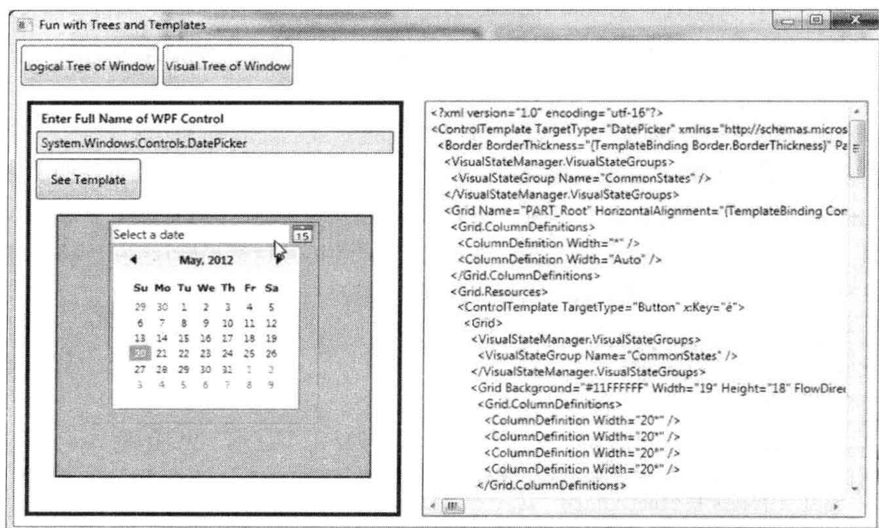


图31-9　在运行时查看ControlTemplate

希望你已经对逻辑树、可视树以及控件模板如何合作有了更好的理解。现在，你可以阅读本章的剩余部分，学习如何构建自定义模板和用户控件。

源代码　TreesAndTemplatesApp项目的源代码位于Chapter 31子目录下。

31.5　使用触发器框架构建自定义控件模板

仅仅使用C#代码就可以为控件创建自定义模板。这时，你需要为一个ControlTemplate对象添加数据，然后将其赋值给控件的Template属性。然而大多数情况下，我们使用XAML定义ControlTemplate的外观，并添加少量代码（也可能是大量代码）来驱动运行时行为。

本章剩余部分将学习如何使用Visual Studio构建自定义模板。同时还将学习WPF触发器框架、Visual State Manager（VSM）以及如何使用动画与最终用户的视觉线索交互。你将看到，仅仅使用Visual Studio构建复杂模板将需要大量录入工作，并显得有点重量级。毫无疑问，产品级模板还是要利用Expression Blend的优势。不过，鉴于本书这个版本不会涵盖Blend的内容，现在是时候撸袖子敲键盘写标记了。

创建一个全新的WPF应用程序ButtonTemplate。在该项目中，我们专注于创建和使用模板的机制，因此主窗体的标记十分简单：

```
<Window x:Class="ButtonTemplate.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Fun with Templates" Height="350" Width="525">
    <StackPanel>
      <Button x:Name="myButton" Width="100" Height="100"
              Click="myButton_Click"/>
    </StackPanel>
  </Window>
```

在Click事件处理程序中，显示一个简单的消息框（通过MessageBox.Show()方法），输出一条确认单击控件的消息。记住，在构建自定义模板时，控件的行为是不变的，外观是可变的。

这时，Button使用默认的模板进行呈现。如上一个示例所述，默认模板是一个位于WPF程序集中的BAML资源。在定义自己的模板时，你实际上是用你创建的可视树替换默认的可视树。更新<Button>元素的定义，使用属性元素语法指定新的模板。该模板使控件显示为圆形：

```
<Button x:Name="myButton" Width="100" Height="100"
        Click="myButton_Click">
  <Button.Template>
    <ControlTemplate>
      <Grid x:Name="controlLayout">
        <Ellipse  x:Name="buttonSurface" Fill = "LightBlue"/>
        <Label x:Name="buttonCaption" VerticalAlignment = "Center"
               HorizontalAlignment = "Center"
               FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
      </Grid>
    </ControlTemplate>
  </Button.Template>
</Button>
```

我们在这里定义了一个模板，包含一个命名的Grid控件，该Grid内部为命名的Ellipse和Label。由于没有定义Grid的行和列，每个子控件都将以内容为中心堆叠在前一个控件的上方。现在，运行应用程序，你会发现只有鼠标在Ellipse边界内时，Click事件才会触发（即在圆形边缘，正方形四个角的地方不会触发）！这是WPF模板架构中一个非常强大的特性：你不必重新计算命中测试、边界检查或其他底层细节。因此，即使你的模板使用Polygon对象呈现不规则的几何图形，你也可以完全放心，命中测试的细节是与控件的形状相关的，而不是其外围较大的矩形边界。

31.5.1　模板资源

现在，模板内嵌于特定的Button控件，这限制了重用。在理想情况下，我们应该将模板放置于资源字典中，这样就可以在多个项目中使用这个圆形的按钮，或者至少将它移动到应用程序资源容器中，从而在当前项目内进行重用。我们使用Visual Studio将本地Button资源移动到应用程序级别。首先，在Properties编辑器中找到Button的Template属性。单击白色方形小图标，选择Convert to New Resource...（如图31-10所示）。

在弹出的对话框中，将新模板取名为RoundButtonTemplate，保存在Application级别（也就是保存在App.xaml中，如图31-11所示）。

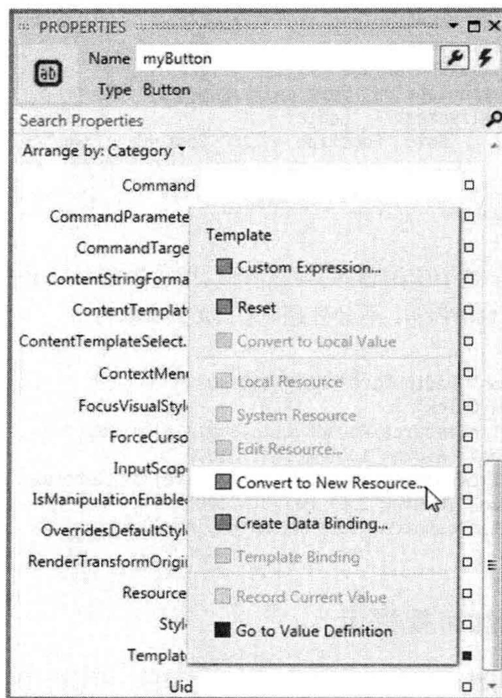


图31-10 提取本地资源

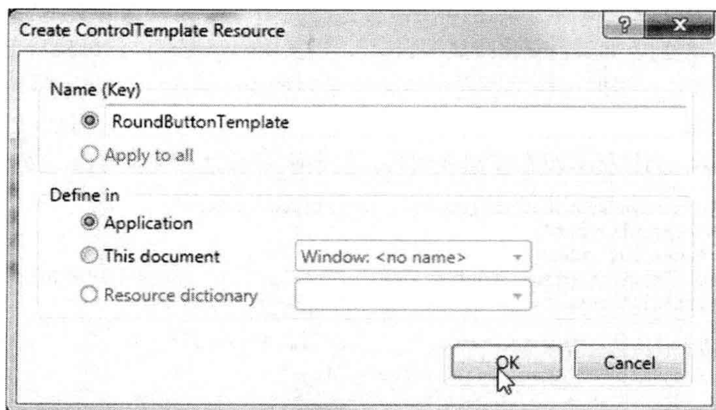


图31-11 将资源放置于App.xaml

这时，在Application对象标记中可以找到如下数据：

```
<Application x:Class="ButtonTemplate.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
<Application.Resources>
```

```
< ControlTemplate x:Key="RoundButtonTemplate" TargetType="{x:Type Button}>
      <Grid x:Name="controlLayout">
        <Ellipse  x:Name="buttonSurface" Fill = "LightBlue"/>
        <Label x:Name="buttonCaption" VerticalAlignment = "Center"
               HorizontalAlignment = "Center"
               FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
      </Grid>
    </ControlTemplate>
  </Application.Resources>
</Application>
```

现在，由于该资源可用于整个应用程序，我们可以用它来定义任意多个圆形按钮。为了测试，继续创建两个使用该模板的Button控件（不必处理它们的Click事件）。

```
<StackPanel>
  <Button x:Name="myButton" Width="100" Height="100"
          Click="myButton_Click"
          Template="{StaticResource RoundButtonTemplate}"></Button>
  <Button x:Name="myButton2" Width="100" Height="100"
          Template="{StaticResource RoundButtonTemplate}"></Button>
  <Button x:Name="myButton3" Width="100" Height="100"
          Template="{StaticResource RoundButtonTemplate}"></Button>
</StackPanel>
```

31.5.2　使用触发器添加可视提示

定义自定义模板时，所有默认模板的可视提示也随之移除。例如，默认的按钮模板包含一些标记，在某个UI事件触发时（如获取焦点、鼠标单击、激活或禁用等），这些标记将通知控件显示什么样的外观。用户早已习惯了这些可视提示，因为它使控件有一种触觉响应的感觉。但是，RoundButtonTemplate没有定义这些标记，因此无论鼠标怎么移动，控件的外观都不会发生任何改变。理想情况下，当控件被单击时应该有细小的差别（如颜色变化或显示阴影），以使用户知道控件的可视状态发生了改变。

在WPF首次发布时，添加这种可视提示的方法是向模板中添加一些触发器，当触发器的条件为true时，将改变对象属性的值或开始一个动画演示图板（或两者同时）。用下面的标记更新RoundButtonTemplate，当鼠标悬浮在按钮表面时，按钮的背景色将变为蓝色，前景色将变为黄色：

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button" >
  <Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="LightBlue" />
    <Label x:Name="buttonCaption" Content="OK!" FontSize="20" FontWeight="Bold"
           HorizontalAlignment="Center" VerticalAlignment="Center" />
  </Grid>
    <ControlTemplate.Triggers>
      <Trigger Property = "IsMouseOver" Value = "True">
        <Setter TargetName = "buttonSurface" Property = "Fill" Value = "Blue"/>
        <Setter TargetName = "buttonCaption" Property = "Foreground"
                Value = "Yellow"/>
      </Trigger>
    </ControlTemplate.Triggers>
  </ControlTemplate>
```

再次运行程序，当鼠标进入或离开Ellipse区域时，颜色将发生改变。下面是另一个触发器，它将在鼠标按下时缩小Grid（及其所有子元素）的尺寸。将以下代码添加到<ControlTemplate.Triggers>集合中：

```
<Trigger Property = "IsPressed" Value="True">
  <Setter TargetName="controlLayout"
          Property="RenderTransformOrigin" Value="0.5,0.5"/>
  <Setter TargetName="controlLayout" Property="RenderTransform">
    <Setter.Value>
      <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
    </Setter.Value>
  </Setter>
</Trigger>
```

31.5.3 {TemplateBinding}标记扩展的作用

我们的模板目前只能用于Button控件，但当你在<Button>元素中添加一些属性，使模板能以独特的方式呈现时，按钮不会发生任何变化。例如，现在Ellipse的Fill属性硬编码为蓝色，Label的Content永久设置为字符串“OK”。如果你需要按钮能显示不同的颜色和文本，可以在主窗体中定义如下的按钮：

```
<StackPanel>
  <Button x:Name="myButton" Width="100" Height="100"
          Background="Red" Content="Howdy!"
          Click="myButton_Click"
          Template="{StaticResource RoundButtonTemplate}" />
  <Button x:Name="myButton2" Width="100" Height="100"
          Background="LightGreen" Content="Cancel!"
          Template="{StaticResource RoundButtonTemplate}" />
  <Button x:Name="myButton3" Width="100" Height="100"
          Background="Yellow" Content="Format"
          Template="{StaticResource RoundButtonTemplate}" />
</StackPanel>
```

尽管每个Button都设置了不同的Background和Content值，但最后我们得到的仍然是3个蓝色的显示“OK”的按钮。原因在于控件（Button）的属性并没有与模板的项一一对应（如Ellipse的Fill属性）。同样，尽管Label包含Content属性，但是在<Button>中定义的值没有自动映射到模板的内部子元素上。

在构建模板时，可以使用{TemplateBinding}标记扩展来解决这个问题。它可以使用模板来获取控件定义的属性设置，并用这些设置来为模板中的相应属性赋值。以下是重写的RoundButtonTemplate，它使用标记扩展将Button的Background属性映射到Ellipse的Fill属性，同时还将Button的Content传递给Label的Content：

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button">
  <Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
    <Label x:Name="buttonCaption" Content="{TemplateBinding Content}"
           FontSize="20" FontWeight="Bold"
           HorizontalAlignment="Center" VerticalAlignment="Center" />
  </Grid>
  <ControlTemplate.Triggers>
   ...
  </ControlTemplate.Triggers>
</ControlTemplate>
```

这样修改之后就可以创建不同颜色和不同文本值的按钮了（如图31-12所示）。

31

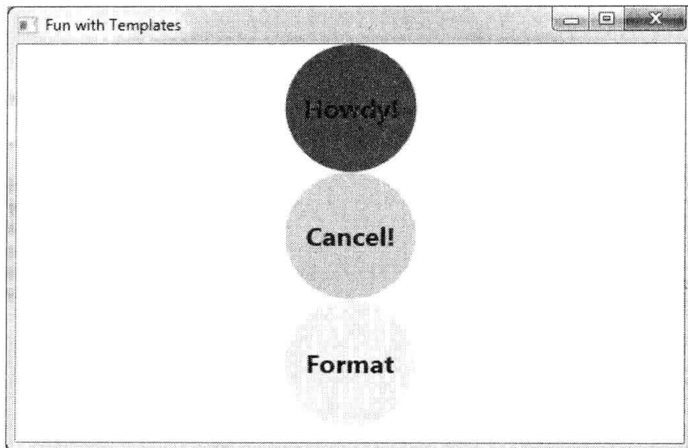


图31-12　模板绑定可以将值传递给内部控件

31.5.4　ContentPresenter的作用

在设计模板时，我们使用一个Label来显示控件的文本值。如前面的Button、Label为其提供了Content属性。因此，使用{TemplateBinding}定义的按钮，其内容可以比简单的字符串要复杂得多。例如：

```
<Button x:Name="myButton4" Width="100" Height="100" Background="Yellow"
        Template="{StaticResource RoundButtonTemplate}">
  <Button.Content>
    <ListBox Height="50" Width="75">
      <ListBoxItem>Hello</ListBoxItem>
      <ListBoxItem>Hello</ListBoxItem>
      <ListBoxItem>Hello</ListBoxItem>
    </ListBox>
  </Button.Content>
</Button>
```

对于如此特殊的控件，模板仍可以运行良好。但是，如果要对一个没有Content属性的模板成员传递复杂内容时应该怎么办呢？这时要在模板中定义通用的内容显示区域，可以使用ContentPresenter类而不能使用特定类型的控件（Label或TextBlock）。尽管在本例中没有必要这么做，但我们更新了标记，用来演示如何使用ContentPresenter构建自定义模板，使那些应用该模板的控件能够显示其Content属性的值：

```
<!-- 该按钮模板将显示所有设置在宿主按钮Content上的内容-->
<ControlTemplate x:Key="NewRoundButton" TargetType="Button">
  <Grid>
    <Ellipse Fill="{TemplateBinding Background}"/>
    <ContentPresenter HorizontalAlignment="Center"
                      VerticalAlignment="Center"/>
  </Grid>
</ControlTemplate>
```

31.5.5 融合模板和样式

目前为止，我们的模板只简单定义了Button控件的基本外观。建立控件基本属性的过程（内容、字体大小、字体粗细等），是由Button本身负责的：

```
<!--需为Button设置基本属性的值，而不是模板-->
<Button x:Name ="myButton" Foreground ="Black" FontSize ="20" FontWeight ="Bold"
        Template ="{StaticResource RoundButtonTemplate}" Click ="myButton_Click"/>
```

如果愿意，你也可以在模板中建立这些值。这样就可以有效地创建一个默认的外观。你可能已经意识到这应该是WPF样式的工作。在为基本属性设置构建样式时，可以在样式内部定义模板。以下是App.xaml中修改后的应用程序资源，它的key已经修改为RoundButtonStyle：

```
<!-- 包含模板的样式 -->
<Style x:Key ="RoundButtonStyle" TargetType ="Button">
  <Setter Property ="Foreground" Value ="Black"/>
  <Setter Property ="FontSize" Value ="14"/>
  <Setter Property ="FontWeight" Value ="Bold"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="100"/>
  <!-- 以下是模板 -->
  <Setter Property ="Template">
    <Setter.Value>
      <ControlTemplate TargetType ="Button">
        <Grid x:Name="controlLayout">
          <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
          <Label x:Name="buttonCaption" Content ="{TemplateBinding Content}"
                 HorizontalAlignment="Center" VerticalAlignment="Center" />
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property = "IsMouseOver" Value = "True">
            <Setter TargetName = "buttonSurface" Property = "Fill" Value = "Blue"/>
            <Setter TargetName = "buttonCaption" Property = "Foreground" Value = "Yellow"/>
          </Trigger>
          <Trigger Property = "IsPressed" Value="True">
            <Setter TargetName="controlLayout"
                    Property="RenderTransformOrigin" Value="0.5,0.5"/>
            <Setter TargetName="controlLayout" Property="RenderTransform">
              <Setter.Value>
                <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
              </Setter.Value>
            </Setter>
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

这样，就可以像下面这样设置按钮控件的Style属性：

```
<Button x:Name="myButton" Background="Red" Content="Howdy!"
        Click="myButton_Click" Style="{StaticResource RoundButtonStyle}"/>
```

尽管按钮的外观和行为是一样的，但将模板内嵌到样式中的好处是，可以为常用属性提供默认值。

以上就是如何使用Visual Studio 和触发器框架构建控件的自定义模板。虽然关于WPF API还有很多内容没有介绍，但我们已经为以后的学习打下了一个良好的基础

源代码　ButtonTemplate项目的源代码位于Chapter 31子目录下。

31.6　小结

本章介绍了很多WPF相关的话题，都是关于自定义用户控件的。我们先学习了WPF是如何使用传统的.NET编程基元（如属性和事件）来定义翻转的。如你所见，依赖属性可以用来构建集成了WPF服务（动画、数据绑定、样式等）的属性。同样，路由事件提供了一种将事件在标记树中向上或向下传递的方式。

这之后我们学习了逻辑树和可视树之间的关系。逻辑树基本上与描述WPF根元素的标记是一一对应的。逻辑树的背后是更深层次的可视树，它包含详细的呈现指令。

接下来解释了默认模板的作用。我们构建自定义模板时，实际上就是用自定义的实现来替换控件全部（或部分）的可视树。

Part 8 第八部分

ASP.NET Web Form

本部分内容

第32章 ASP.NET Web Form

到目前为止，本书中所有的例子都是基于控制台和GUI的。在本书余下来的几章中，我们将探究.NET平台如何助力ASP.NET技术构建基于浏览器的表现层。首先，我们将快速浏览一下许多以Web为中心的关键概念（HTTP、HTML、客户端脚本和回送）、商业Web服务器（IIS）的作用以及ASP.NET Development Web Server。

随着以Web为主线的铺展，本章的剩余部分将主要讨论ASP.NET表单编程模型（包括单页面和代码隐藏页面）的结构，并且研究Page基类的功能。本章还会介绍ASP.NET Web控件的作用、ASP.NET网站的目录结构以及如何通过XML指令使用web.config文件来控制网站的运行时操作。

说明 如果希望加载本书可下载源代码里的ASP.NET网站项目，可启动Visual Studio，选择File→Open→Web Site...菜单选项。在弹出的对话框中，单击File System按钮（位于左侧），选择包含Web项目文件的文件夹。这将在Visual Studio IDE中加载当前Web应用程序的所有内容。

32.1 HTTP 的作用

Web应用程序与传统的桌面应用程序有很多不同。第一个明显的不同之处，就是产品级的Web应用程序将总是包括至少两台联网的机器：一台承载网站，另一台在Web浏览器中查看数据。当然，在开发过程中采用单独一台机器同时担当基于浏览器的客户端和Web服务器也是完全可能的。因此，这些机器必须遵从一个特定的联网协议以决定如何发送和接收数据。连接计算机的联网协议就是HTTP（超文本传输协议）。

32.1.1 HTTP请求/响应循环

当一台客户端计算机运行一个Web浏览器（例如Google Chrome、Opera、Mozilla Firefox、Apple Safari或者微软Internet Explorer）时，就会建立一个HTTP请求以访问远程服务器上的特定资源（通常是一个网页）。HTTP就是一种以文本为基础的协议，它建立在一个标准的请求/响应范型上。例如，如果你输入了http://www.facebook.com，浏览器软件将使用名为DNS（域名服务）的Web技术，DNS将被注册的URL转换成一个IP地址。此刻，浏览器打开一个套接字连接（通常经由端口80，这是不安全连接），并且向目标站点发送HTTP请求。

Web服务器收到外来的HTTP请求时，将会解析任何客户端提供的输入值（例如文本框中包含的值、

复选框或列表框等）的逻辑，以格式化一个正确的HTTP响应。Web程序员可以采用任何服务器端技术（PHP、ASP.NET、JSP等）动态地产生一些内容以加入到HTTP响应中。此刻，客户端浏览器呈现出从Web服务器发送来的HTML页面。图32-1阐明了基本的HTTP请求/响应循环。

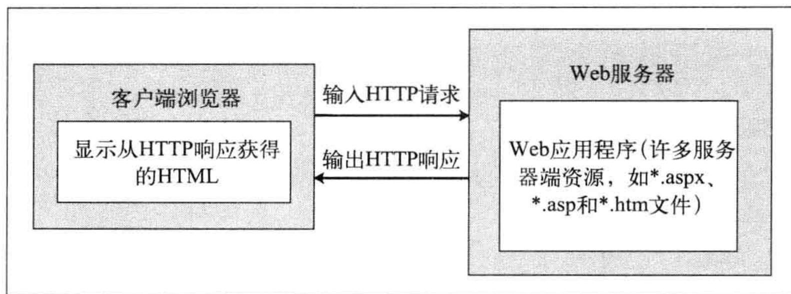


图32-1　HTTP请求/响应循环

32.1.2　HTTP是无状态协议

另一个显著不同于桌面程序的Web开发就是，HTTP是一种本质上无状态的联网协议。只要Web服务器提交给客户端一个响应，先前的所有交互都将被遗忘。对于传统的桌面应用程序来说则不是如此，执行的结果通常会保留到用户关闭应用程序。

因此，作为一名Web开发者，由你决定使用具体步骤“记录”当前登录站点的客户信息（例如，在购物车中的物品、信用卡账号和家庭住址等）。在第34章中将看到，ASP.NET提供了大量处理状态的方法，如使用会话变量、cookie、应用程序缓存等技术以及ASP.NET配置管理API。

32.2　Web应用程序和Web服务器

Web应用程序可理解成为各种文件（*.htm、*.aspx、图像文件、基于XML的文件数据等）和存储在指定Web服务器上的一套特定目录集内的相关组件（例如.NET代码库）的集合。正如第34章将介绍的，ASP.NET Web应用程序具有特定的生命周期，并且提供许多事件（例如初始化事件和关闭事件）以让我们挂钩并进行自定义的处理。

Web服务器就是一个负责承载Web应用程序的软件产品。通常情况下，它会提供许多相关的服务，例如集成安全、FTP（文件传输协议）支持、邮件交换服务等。IIS（Internet信息服务）是微软企业级Web服务器产品，它提供了对ASP.NET Web应用程序的内在支持。

假定你已经在工作站上正确安装了IIS，这样就能通过双击Internet Information Service从Administrative Tools文件夹（位于Control Panal文件夹）中与IIS进行交互。图32-2显示了IIS中的Default Web Site节点，这里可以配置大多数细节（如果运行的是较早版本的IIS，界面会有所不同）。

32.2.1　IIS虚拟目录的作用

一个IIS安装能承载许多Web应用程序，每一个虚拟目录中驻留一个Web应用程序。在本地硬盘中，

每一个虚拟目录都被映射到一个物理目录。例如，如果你创建了一个新的名为CarsAreUs的虚拟目录，外部世界就能够通过使用一个像http://www.MyDomain.com/CarsAresUs（假设站点的IP地址已经注册）这样的URL映射到这个站点。虚拟目录映射到一个包含了CarAreUs Web应用程序内容的Web服务器上的物理根目录。

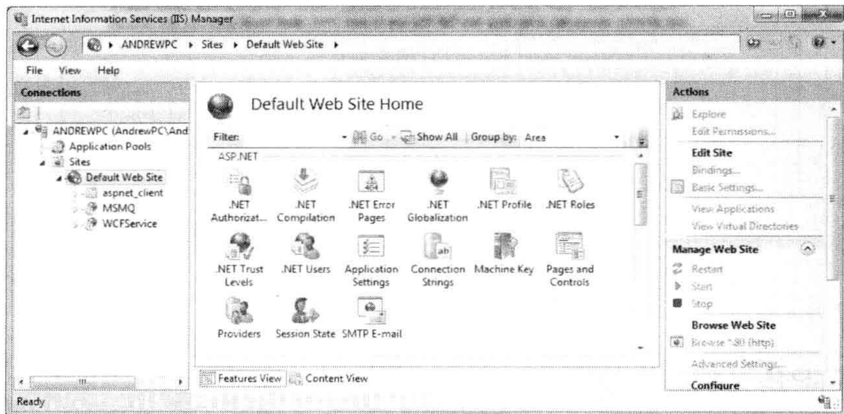


图32-2　可以使用IIS程序配置微软IIS的运行时行为

在本章后面我们会看到，当使用Visual Studio创建ASP.NET Web应用程序时，可以使用IDE来为当前的网站自动生成新的虚拟目录。然而如果需要的话，当然也可以通过右击IIS的 Default Web Site节点，然后从弹出的上下文菜单中选择AddVirtual Directory来创建新的虚拟目录。

32.2.2　ASP.NET Development Web Server

在早期的.NET版本中，ASP.NET的开发者需要在开发和测试Web应用程序的时候使用IIS虚拟目录。在很多情况下，这和设置IIS的团队有紧密的依赖性，并带来不必要的复杂性，更不要说很多网络管理员不喜欢在每一个开发者的机器上安装IIS。

好在我们现在可以选择使用一个叫做ASP.NET Development Web Server的轻量级Web服务器。这个工具允许开发者在IIS的范围外承载一个ASP.NET Web应用程序。使用这个工具，你能从计算机的任何目录建立和测试网页。这对团队开发非常有帮助，对于在不支持IIS安装的Windows版本上建立ASP.NET Web应用程序也是非常有帮助的。

本书中的大多数示例都会（通过正确的Visual Studio项目选项）使用ASP.NET Development Web Server，而不是在IIS虚拟目录下承载Web内容。虽然这个方式可以简化Web应用程序的开发，但是要知道这个Web服务器往往不会承载产品级别的Web应用程序。这只用于开发和测试目的。在Web应用程序完成之后，我们的网站还是需要复制到IIS虚拟目录中。

说明　Visual Studio提供了内嵌的工具，可以将本地Web应用复制到产品级的Web服务器，只需要点击一两个按钮即可。要开始这一过程，需要在Visual Studio的Solution Explorer中选择Web项目，然后点击Copy Web Site按钮。这时就可以选择想要部署的目标位置了。

32.3 HTML 的作用

配置好一个承载Web应用程序的目录并且已经选择Web服务器作为主机之后，就需要创建内容。回想一下，Web应用程序只是构成站点功能的文件集。当然，这些文件中的大部分包含用HTML定义的符合语法的标记。HTML是一种标准标记语言，用来描述文本、图像、外部链接和各种HTML控件是如何被客户端浏览器呈现的。

虽说现在的很多IDE（包括Visual Studio）和Web开发平台（例如ASP.NET）都会自动生成大量的HTML，但在ASP.NET平台上工作时，拥有一些HTML知识是很有益的。

说明 回忆一下第2章，微软在Express产品家族中发布了很多免费的IDE（如Visual C# Express）。如果你对Web开发感兴趣的话，可以下载Visual Web Developer Express。这个免费的IDE专门用于构建ASP.NET Web应用程序（可使用C#或VB）。

虽然这部分内容肯定不会（也不可能）覆盖HTML的所有方面，但还是让我们来看一下基本概念。这将有助于你更好地理解ASP.NET Web表单编程模型为你生成的标记。

32.3.1 HTML文档结构

HTML文件是由描述网页外观的一组标签组成的。HTML文档的基本结构一般保持一致。例如，*.htm文件（或者*.html文件）开始于`<html>`标签和结束于`</html>`标签，通常定义`<body>`部分，等等。

为说明一些HTML基础知识，打开Visual Studio，使用File→New→File菜单选项插入一个空的HTML页面文件（注意此时并没有构建Web项目，而只是创建了一个用于编辑的空的HTML文件），然后将这个文件保存到硬盘中方便的位置，名字为default.htm。可以看到如下所示的初始标记（具体的HTML可能会因Visual Studio的配置不同而不同）：

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml" >
  <head>
    <meta charset="utf-8" />
    <title>Untitled Page</title>
  </head>
  <body>

  </body>
</html>
```

首先，注意HTML文件一开始的`DOCTYPE`处理指令。它和开放的`<html>`标签表明包含的HTML标签应该以HTML 5.0标准进行验证。传统的HTML在语法上非常松散。除了大小写问题外，可以定义一个并没有对应结束标签（这里就是`</br>`）的开始元素（如换行`<br>`）等。HTML 5.0标准是一个W3C规范，它为标记增加了很多新的特性。

32

说明 在默认情况下，Visual Studio按HTML 5.0过渡性验证方案来验证所有的HTML文档。简而言之，HTML 5.0验证方案用于确保标记和某个标准同步。如果你希望指定其他验证方案的话，（如HTML 4.0）激活Tools→Options对话框，展开Text Editor节点，在HTML下选择Validation节点。如果不希望看到验证错误，只要不选中Show Errors复选框即可。

被定义的绝大部分实际内容都位于<body>范围内。为了对页面略做修饰，按如下代码所示为页面定义一个标题：

```
<head>
  <title>This is my simple web page</title>
</head>
```

正如你所料，<title>标签用来指定文本字符串，这些文本字符串应该放在承载Web浏览器的标题栏里。

32.3.2 HTML表单的作用

HTML表单就是一组命名了的用于收集用户输入信息的有关UI元素。不要把HTML表单与给定浏览器的完整显示区域混淆。事实上，HTML表单更多的是在<form>和</form>标签中放置部件的逻辑分组。例如：

```
<!DOCTYPE HTML>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>This is my simple web page</title>
</head>
<body>
  <form id="defaultPage">
    <!-- 在此插入Web用户界面内容 -->
  </form>
</body>
</html>
```

这个表单指定了"defaultPage"的id。开始标签<form>提供了一个action特性，这个特性指定提交表单数据的URL和数据自身传输的方法（POST或GET）。目前，看一下能放在HTML表单中的项目分类（除了简单的文本以外）。

32.3.3 Visual Studio HTML设计器工具

Visual Studio在工具箱上提供了一个HTML选项卡，可选择一个希望放在HTML设计器中的HTML控件（如图32-3所示）。

说明 在使用Web表单编程模型构建ASP.NET Web页面时，我们通常不使用这些HTML控件来创建用户界面。我们使用的是ASP.NET Web控件，它们可以呈现正确的HTML。本章后面将会学习Web控件的作用。

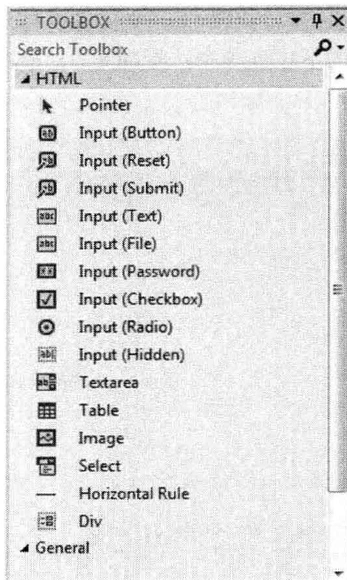


图32-3 工具箱的HTML选项卡

和构建WPF应用程序的过程相似，这些HTML控件可以拖放到HTML设计器界面或直接拖放到HTML标记上。如果单击HTML编辑器底部的Split按钮，HTML编辑器底部会显示HTML可视化布局，而上面一部分会显示相关的标记。这个编辑器的另外一个优势是，在我们选择标记或HTML UI元素的时候，对应的表示会突出显示。（如图32-4所示）。

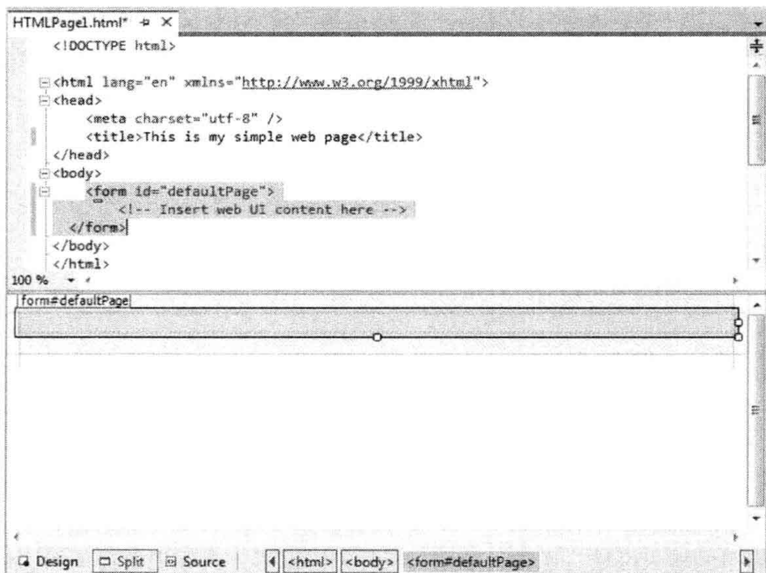


图32-4 Visual Studio HTML编辑器

Visual Studio也允许使用Properties窗口编辑*.htm文件的总体界面外观或者<form>中给定的HTML控件。例如，如果从Properties窗口的下拉列表中选择DOCUMENT，你就能配置HTML页面的各个方面（如图32-5所示）。

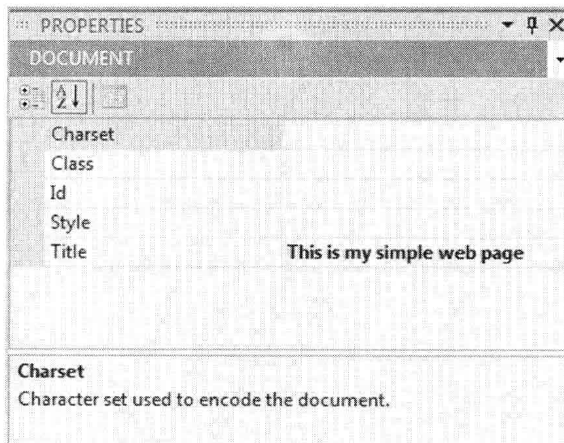


图32-5　通过Visual Studio Properties窗口配置HTML标记

在使用Properties窗口配置Web页面时，IDE将相应地更新HTML。在阅读本书剩余章节的时候，你可以随意使用IDE来帮助编辑HTML页面。

32.3.4　构建HTML表单

现在，更新初始文件中的<body>，以显示一些提示用户输入信息的文本。注意，可以通过在HTML设计器上直接输入或格式化文本内容。下面使用<h1>标签设置标题宽度，<p>设置段落块，<i>设置斜体文本：

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>This is my simple web page</title>
</head>
<body>
  <!-- 提示用户进行输入 -->
  <h1>Simple HTML Page</h1>
  <p>
    <br/>
    <i>Please enter a message</i>.
  </p>

  <form id="defaultPage">
  </form>

</body>
</html>
```

现在来创建表单的输入区域。通常，使用id特性（以编程方式标识条目）和type特性（用于指定将哪些感兴趣的输入控件放置在<form>声明中）来描述每一个HTML控件。依据所使用的UI窗口部件，

将发现针对特定条目的附加特性，这些条目可以使用Properties窗口来修改。

即将构建的UI包括一个文本字段和两个按钮类型。第一个按钮用来运行客户端脚本，另一个按钮用来将表单数据重置为默认值。按如下所示更新HTML表单：

```
<!-- 建立一个表单来获取用户信息 -->
<form id="defaultPage">
  <p>
    Your Message:
    <input id="txtUserMessage" type="text"/></p>
  <p>
    <input id="btnShow" type="button" value="Show!"/>
    <input id="btnReset" type="reset" value="Reset"/>
  </p>
</form>
```

注意，你已经给每个控件（txtUserMessage、btnShow和btnReset）指定了相应的id，并且注意每一个输入项目都有一个名为type的额外特性，表示这些输入控件是UI项目，并且能自动将所有字段的值设为初始值（type="reset"），接收文本输入（type="text"）或与客户端按钮功能类似但并不向Web服务器回发数据（type="button"）。

保存文件，右击并选择View in Browser菜单选项。图32-6在Firefox浏览器中显示了当前页面。

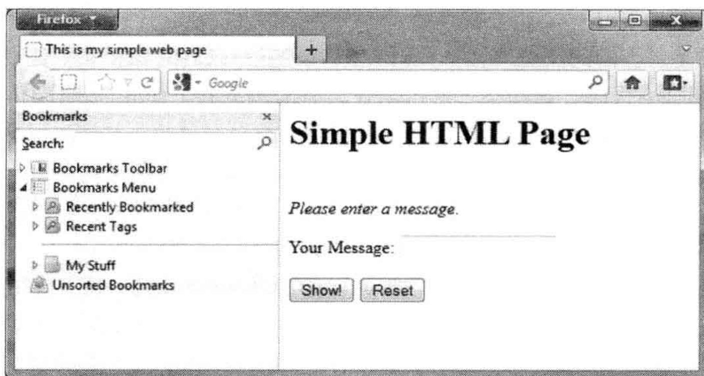


图32-6 简单的HTML页面

说明 当对一个HTML文件选择View in Browser选项时，Visual Studio将自动启动ASP.NET Development Web Server来承载内容。

32.4 客户端脚本的作用

除了GUI元素外，一个给定的*.htm文件可以包含脚本代码块，该代码块将由请求浏览器处理。使用客户端脚本的两个主要原因如下：

- 在回传到Web服务器之前验证用户输入；
- 与目标浏览器的DOM（文档对象模型）交互。

关于第一点，要理解的是，Web应用程序天生的麻烦就是，需要经常制造往返过程（即回传）到服务器来更新呈现到浏览器中的HTML。既然回传不可避免，在通信过程中总是要考虑使往返过程最小化。节省回传的一项技术是，在向Web服务器提交表单数据前，使用客户端脚本来验证用户输入。如果发现了错误（例如在必填字段上没有指定数据），就能够提示用户这个错误，而不至于把错误值回传给Web服务器。（毕竟对于用户来讲，在一个低速连接上回传地址输入错误的说明太令人讨厌了！）

说明　要知道即使进行客户端验证（为了提高响应时间），在Web服务器还是应该再验证一次。这将确保数据在网络上传递时没有被篡改。在第33章中会介绍，ASP.NET验证控件会自动进行客户端和服务器端的验证。

客户端脚本也能够用来与浏览器自身的底层对象模型（DOM）交互。多数商业浏览器都公开一组能够用来控制浏览器应该怎样运作的对象。

浏览器解析HTML页面时，会在内存中建立一个对象树，表示Web页面中的所有内容（窗体、输入控件等）。浏览器提供了一个叫做DOM的API，它公开了对象树并允许以编程方式修改其内容。例如，你可以编写在浏览器中执行的JavaScript，用来获取指定控件的值、更改控件的颜色、在页面中动态添加控件等。

一个主要的麻烦是，不同的浏览器往往公开相似*但却不一致*的对象模型。因此，假如你写了一段与DOM交互的客户端脚本代码，它却不一定可以在所有的浏览器上做相同的工作。

ASP.NET提供了`HttpRequest.Browser`属性，它允许你在运行时确定发送当前请求的浏览器的功能。你可以使用这些信息来决定如何以最优的方式回发HTTP响应。但除非实现自定义控件，否则你没有必要担心这些，因为ASP.NET中所有标准的Web控件都能自动根据浏览器类型以适当的方式进行呈现。这个了不起的功能称为*适应性呈现*，它为所有标准的ASP.NET控件而实现。

编写客户端脚本代码的脚本语言有很多，最流行的当属JavaScript。要十分清楚的是，JavaScript绝不是Java语言的另一种形式或一个子集。虽然JavaScript和Java有一些相似的语法，但是JavaScript不是一个完全的面向对象编程语言，因此能力远远不及Java。好消息是当前所有的Web浏览器都支持JavaScript，使得对于客户端脚本逻辑来说，JavaScript成了一个自然而然的选择。

客户端脚本示例

为了说明客户端脚本的作用，让我们先来检查怎样截获从客户端HTML GUI部件发送的事件。为了捕获这个按钮的`click`事件，更新`btnShow`控件的定义，使其支持 `onclick`特性，该特性的值为叫做`btnShow_onclick()`的JavaScript方法。

```
<input id="btnShow" type="button" value="Show!"
       onclick="return btnShow_onclick()" />
```

下面，直接在`<head>`元素后面添加如下所示的JavaScript函数，当用户单击该按钮时将调用该函数。使用`alert()`方法通过`value`属性来显示一个客户端消息对话框（其中包括文本框中的值）：

```
<script lang="javascript" type="text/javascript">
// <![CDATA[
  function btnShow_onclick() {
```

```
      alert(txtUserMessage.value);
    }
  // ]]>
  </script>
```

注意，脚本块已经被包装在CDATA中。原因很简单，如果你的页面在一个不支持JavaScript的浏览器上显示，那么这段代码将被当做注释块处理并被忽略。当然，你的页面就缺少这个功能，不过好的一面是，在被浏览器呈现时，你的页面不会出问题。再次在浏览器中查看该页面，输入一些内容，可以看到它会在客户端信息框中弹出（如图32-7所示）。

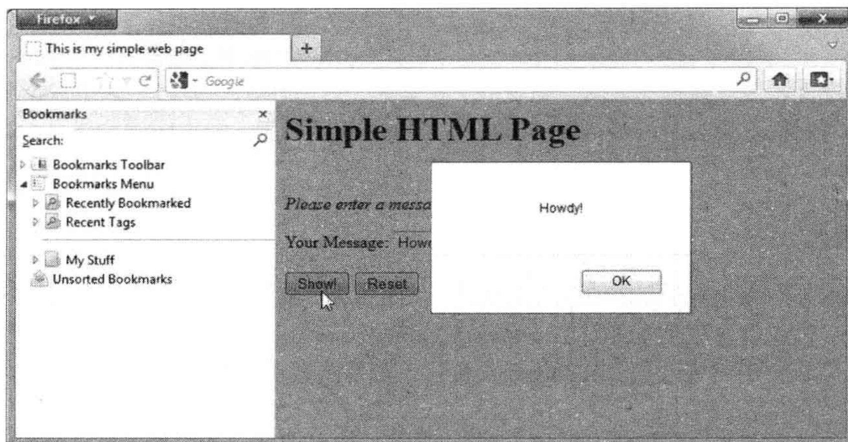


图32-7 调用客户端JavaScript函数

同样，单击Reset按钮，文本区域内的所有数据都被清空，因为该按钮在创建时指定了type="reset"。

32.5 回发到 Web 服务器

这个简单HTML页面的所有功能都在浏览器中执行。而真正的Web页面需要向Web服务器中的某个资源进行回发，同时传送所有输入数据。服务器资源接收这些数据，并用它们来构建适当的动态生成的HTTP响应。

在开始标记<form>中，action特性指定了输入表单数据的接收者。接收者可能为邮件服务器、Web服务器上的其他HTML文件、标准的ASP文件、ASP.NET Web页面等。

除了action特性，你还可能会创建一个*提交按钮*，该按钮在单击时会将表单数据通过HTTP请求传送给Web应用程序。我们没有必要在本例中这么做，但下面这个更新后的default.htm在开始标签<form>中指定了以下特性：

```
<form id="defaultPage"
      action="http://localhost/Cars/ClassicAspPage.asp" method="GET">
  <input id="btnPostBack" type="submit" value="Post to Server!"/>
...
</form>
```

单击该表单中的提交按钮，表单数据将被发送给URL所指定的ClassicAspPage.asp。如果将传送方式指定为method="GET"，表单数据将以&符号分隔的名称/值对的形式附加到查询字符串中。你肯定在浏览器中看到过这种类型的数据，如：

```
http://www.google.com/search?hl=en&source=hp&q=vikings&cts=1264370773666&aq=
f&aql=&aqi=g1g-z1g1g-z1g1g-z1g4&oq=
```

将表单数据传送给Web服务器的另一种方法是指定method="POST"，如下所示：

```
<form id="defaultPage"
      action="http://localhost/Cars/ClassicAspPage.asp" method = "POST">
...
</form>
```

在这种情况下，表单数据不会附加到查询字符串中。使用POST方式时，表单数据对于外部世界来说不是直接可见的。更重要的是，POST数据没有字符长度限制，而许多浏览器对于GET查询是有字符限制的。

ASP.NET回发

在使用Web 表单构建基于ASP.NET的网站时，框架将为我们管理回发机制。用ASP.NET构建网站的好处之一是，编程模型在标准的HTTP request/response协议的上层放置了一个事件驱动系统。因此，你不必手工设置action特性和定义HTML提交按钮，只需要简单地使用标准的C#语法处理ASP.NET Web控件的事件就可以了。

使用这种事件驱动模型，你可以使用大量控件触发对Web服务器的回发。单击单选按钮、复选框中的某项、日历控件中的某天等，都可以向Web服务器回发数据。你只需要简单地处理正确的事件，ASP.NET运行时将自动回传正确的HTML数据。

源代码　SimpleWebPage网站的源代码位于Chapter 32子目录下。

32.6　ASP.NET API 概览

至此，我们回顾了经典的Web应用程序开发，你已经准备好开始学习ASP.NET了。正如你所预料的那样，每个.NET平台的版本都向Web编程API中添加了很多功能，.NET 4.5亦是如此。不管你面向的是哪个.NET版本，对于ASP.NET Web应用来说下面的特性都是通用的。

- ASP.NET提供了名为代码隐藏的模型，允许将表现逻辑（HTML）从业务逻辑（C#代码）中分离出来。
- ASP.NET页面使用.NET编程语言而不是服务器端脚本语言编码。这个代码文件可以编译为有效的.NET *.dll程序集，可以带来更快的执行速度。
- ASP.NET Web控件可用来构建Web UI，其模型与桌面窗口应用非常类似。
- ASP.NET Web应用程序可以使用.NET基础类库中的任何一个程序集，并且用本书介绍的面向对象技术（类、接口、结构、枚举和委托）进行构建。
- 能使用Web应用程序配置文件（web.config）轻松配置ASP.NET Web应用程序。

这里首先要阐述的是，ASP.NET Web页面的UI可使用多个Web控件构建。与典型的HTML控件不同，Web控件在Web服务器上执行，然后将正确的HTML标签回发给HTTP响应。仅此一点，就可以大大减少手工编写HTML的数量。来看一个快速示例，假设在ASP.NET Web页面中定义了如下的ASP.NET Web控件：

```
<asp:Button ID="btnMyButton" runat="server" Text="Button" BorderColor="Blue"
        BorderStyle="Solid" BorderWidth="5px" />
```

稍后我们将学习创建ASP.NET Web控件的细节，现在要注意的是，<asp:Button>控件的很多特性看上去都与WPF示例中的属性十分类似。所有ASP.NET Web控件都是如此。这是因为微软在构建这些Web控件工具包时，特意将其外观设计得很像桌面框架中对应的控件。

现在，如果浏览器调用包含该控件的*.aspx文件，控件将被回发的输出流响应为如下的HTML声明：

```
<input type="submit" name="btnMyButton" value="Button" id="btnMyButton"
      style="border-color:Blue;border-width:5px;border-style:Solid;" />
```

注意Web控件可以在任何浏览器中回发标准的HTML。因此，ASP.NET Web控件并不局限于微软的操作系统家族和IE。任何操作系统或浏览器（包括Apple iPhone或BlackBerry这种手持设备）都可以浏览ASP.NET Web页面。

注意前面列表中提到的一个特性，即ASP.NET Web应用程序将被编译为.NET程序集。因此，Web项目与本书中的其他.NET *.dll没有任何区别。编译的Web应用程序由CIL代码、程序集清单和类型元数据组成。这可以带来巨大的好处，如显著的性能提升、强类型和由CLR托管的能力（如垃圾回收）。

最后，ASP.NET Web应用程序还提供了一种编程模型，它使用*代码文件*将页面标记和相关的C#代码相分离。使用代码文件，你输入的标记将映射到一个完整的对象模型，该对象模型由多个C#代码文件中声明的分部类合并而成。

32.6.1 ASP.NET 2.0 及其后续版本的主要特性

ASP.NET 1.0往正确方向迈了一大步，而ASP.NET 2.0提供了许多额外的特性，使ASP.NET的定位从构建动态网页到构建功能丰富的网站。主要改进如下。

- 引入ASP.NET Development Web Server（也就是说，开发人员无须在开发用的计算机上安装IIS）。
- ASP.NET 2.0含有大量处理复杂情况的新Web控件（导航控件、安全控件、新数据绑定控件等）。
- ASP.NET 2.0支持*母版页*的使用，这允许给一套相关页面附加一个公共的UI框架。
- ASP.NET 2.0支持*主题*，它提供了一个可声明的方式改变整个Web应用程序的界面外观。
- ASP.NET 2.0支持Web*部件*，允许终端用户定制网页的界面外观并存储这些设置以便以后使用。
- ASP.NET 2.0支持基于Web的配置文件和管理工具，以维护web.config文件。

除了ASP.NET Development Web服务器，ASP.NET 2.0带来的另一个巨大改变是引入了*母版页*。如你所知，大多数网站的所有页面都有统一的外观。像www.amazon.com这样的商务网站，每个页面都包含相同的元素，如公共的头部、底部、导航菜单等。

使用母版页，可以建立这些公共的功能，并定义可容纳*.aspx文件的占位符。这样，不需要更改其他*.aspx文件，只需要更改母版页，就可以轻松快速地改变站点的整个外观（改变导航条的位置，更改头部logo等）。

说明　Visual Studio 2010广泛使用了母版页，所有的ASP.NET Web项目都包含默认的母版页。

ASP.NET 2.0还引入了许多新的混合Web控件，包括自动合并了公共安全特性的控件（登录控件、密码恢复控件等）、在一组相关的*.aspx文件顶部放置导航结构的控件，以及更多执行复杂数据绑定操作的控件，它们可以使用多个ASP.NET Web控件生成必要的SQL查询。

32.6.2　ASP.NET 3.5（和.NET 3.5 SP1）的主要特性

.NET 3.5使ASP.NET Web应用程序能够使用LINQ编程模型（.NET 3.5引入的），并添加如下Web相关的特性。

- 支持对ADO.NET实体类的数据绑定（见第23章）。
- 支持ASP.NET Dynamic Data。这是一个类似Ruby on Rails的Web框架，可用来构建数据驱动的Web应用程序。它将数据库中的表编码到ASP.NET Web服务的URI中，表中的数据将自动呈现为HTML。
- 集成了对Ajax风格开发的支持，它实质上支持部分回发，可以快速刷新Web页面的部分内容。

.NET 3.5 SP1中的ASP.NET Dynamic Data项目模板提供了一种新的模型，用于构建由关系型数据库驱动的站点。当然，大多数网站都在一定程度上需要与数据库进行通信，但ASP.NET Dynamic Data项目与ADO.NET Entity Framework仅仅联系在一起，并且专注于数据驱动网站的快速开发（这与Ruby很类似）。

32.6.3　ASP.NET 4.0 和 4.5 的主要特性

.NET 4.0和4.5为微软Web开发平台增加了更多的特性。下面列出了一些关键的Web特性。

- 使用GZIP标准压缩“视图状态”数据。
- 更新了浏览器的定义，以确保ASP.NET页面能正确呈现在新的浏览器和设备上（Google Chrome、Apple iPhone、BlackBerry设备等）。
- 使用CSS定制验证控件的输出结果。
- 新增ASP.NET Chart控件，可构建包含复杂统计或财务分析图表的ASP.NET页面。
- 支持ASP.NET Model View Controller项目模板，它使用MVC模式，降低了应用程序层之间的依赖性。这是网站开发的一种完全不同的方式，与本书所介绍的Web表单编程模型相似性很小。
- 为支持HTML 5.0做出了很多更新。
- 集成了C#和VB中新的异步语言特性。

ASP.NET的这些特性相当有深度（而且这个API还有很多我没有列出的特性）。说实话，如果真的涵盖ASP.NET的所有特性，本书的厚度将增加一倍（也许两倍）。因此，本书剩余部分的目标是介绍ASP.NET中那些经常使用的核心特性。对于没有介绍到的特性，可以参考.NET Framework 4.5 SDK文档。

说明 如果你要全面地学习如何使用ASP.NET构建Web应用程序，建议你买本Adam Freeman和Matthew MacDonald所著的*Pro ASP.NET 4.5 in C#, Fifth Edition*。

32.7 构建单个文件的 ASP.NET 网页

ASP.NET网页能使用两种主要方法进行创建。第一种方法创建一个*.aspx文件，这个*.aspx文件融合了服务器端代码和HTML。使用单文件页面模型，服务器端代码被放置在<script>作用域内，但是代码本身并不是严格意义上的脚本代码（例如VBScript/JavaScript），在<script>块内的代码语句将使用你所选择的.NET语言（C#、Visual Basic等）。

如果你正在构建一个页面，它包含非常少的代码（但确有大量的静态HTML），使用一个单文件页面模型可以更简单些，因为这样你就可以在一个统一的*.aspx文件中看到所有的代码和标记。另外，把过程式代码和HTML标记放在一个*.aspx文件中具有以下几个优点。

- 部署或发送给另一个开发者使用单文件模型编写的页面要简单些。
- 因为文件互相之间不存在依赖，所以单文件页面更易于重命名。
- 因为所有动作在一个文件中发生，所以在源代码控制系统内管理文件更容易。

但单文件模型的缺点是，它与经典的基于COM的ASP一样，都将导致过于复杂的文件（因为UI标记和程序逻辑被隔离在一个地方）。不过，我们的ASP.NET之旅还是将从单文件页面模型开始。

我们的目标是建立一个*.aspx文件，用它来显示AutoLot数据库（在第21章中创建的）的Inventory表（但可以猜到，也可以使用断开连接层或Entity框架）。启动Visual Studio，然后通过File→New→File菜单选项创建一个新的Web Form（如图32-8所示，注意需要在左侧的树形视图中点击Web→C#节点）。

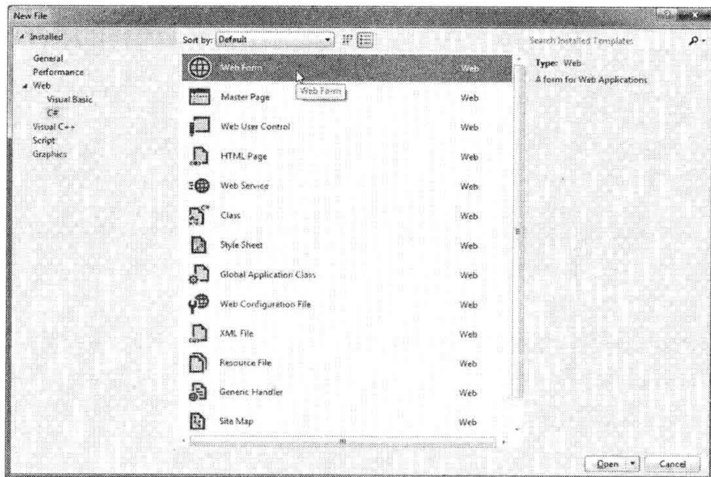


图32-8 创建新的单文件ASP.NET页面

这样做之后，将这个文件保存为Default.aspx，并保存在硬盘上容易找到的新目录下（例如C:\MyCode\SinglePageModel）。

32.7.1　引用AutoLotDAL.dll

接着，使用Windows资源管理器在SinglePageModel文件夹下创建子目录bin。这个特殊的bin子目录是ASP.NET运行时引擎的注册名。在网站根目录下的\bin文件夹中，我们可以部署任何Web应用程序的私有程序集。对于本例，把AutoLotDAL.dll（见第21章）的副本放到C:\MyCode\SinglePageModel\bin文件夹中。

说明　本章稍后会介绍，如果使用Visual Studio创建ASP.NET Web项目，IDE会为我们维护\bin文件夹，并在默认情况下将任何引用复制到私有程序集中。

32.7.2　设计UI

现在使用Visual Studio工具箱，选择Standard选项卡，然后拖放`Button`、`Label`和`GridView`控件到开始元素`form`和结束元素`form`之间的页面设计器上（`GridView`部件能够在工具箱的Data选项卡找到）。随意使用Properties窗口来设置各种各样的可视化属性，然后通过ID特性给每一个Web部件一个合适的名字。图32-9显示了一个可能的设计。（我有意保持外观平淡无奇，以减少生成控件标记的数量，但你可以任意修饰它。）

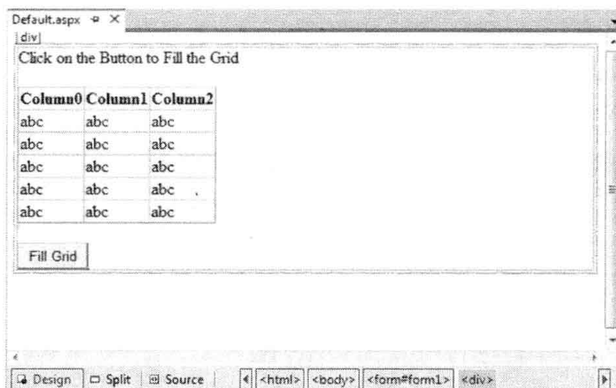


图32-9　Default.aspx GUI

现在，找到页面的`<form>`部分。注意每个Web控件是如何使用`<asp:>`标签定义的。在这个标记前缀之后，是ASP.NET Web控件的名称（`Label`、`GridView`和`Button`）。在给定元素的结束标记之前，你将看到一系列与Properties窗口中的设置相对应的名称/值对，例如：

```
<form id="form1" runat="server">
<div>
  <asp:Label ID="lblInfo" runat="server"
    Text="Click on the Button to Fill the Grid">
  </asp:Label>
  <br />
  <br />
```

```
    <asp:GridView ID="carsGridView" runat="server">
    </asp:GridView>
    <br />
    <asp:Button ID="btnFillData" runat="server" Text="Fill Grid" />
  </div>
  </form>
```

随后第33章将讨论ASP.NET Web控件的全部细节。此前，只要理解Web控件是在Web服务器上处理的对象，并且它们自动地将HTML请求发回到输出的HTTP响应中。除了这个主要的好处之外，ASP.NET Web控件模仿类似桌面的编程模型，因为有和Windows Forms/WPF相同的属性、方法和事件的名称。

32.7.3 添加数据访问逻辑

现在切换到定义者，使用Visual Studio Properties窗口（通过闪电图标）来处理Button类型的Click事件（就像本章回顾HTML时所做的一样）。此后你将发现Button的定义已经得到更新，为Click事件处理程序名称分配了OnClick特性：

```
<asp:Button ID="btnFillData" runat="server"
    Text="Fill Grid" OnClick="btnFillData_Click"/>
```

现在，在服务器端*.aspx的<script>块中编写Click事件处理程序。注意传入参数必须与System.EventHandler委托的目标完全匹配，这在本书中已经出现多次：

```
<script runat="server">
  protected void btnFillData_Click(object sender, EventArgs args)
  {
  }
</script>
```

下一步就是使用AutoLotDAL.dll程序集的功能来填充GridView。我们必须使用<%@ Import %>指令来指定我们使用的AutoLotConnectedLayer命名空间。

说明 只有在使用单文件代码模型构建页面时，你才需要使用<%@ Import %>指令。如果使用默认的代码文件方法，那么只需要简单地使用C# using关键字，就可以在代码文件中引用命名空间。<%@ Assembly %>指令也是如此。

此外，我们需要通过<%@ Assembly %>指令（稍后会介绍有关指令的更多细节）通知ASP.NET运行库，这个单文件页面引用了AutoLotDAL.dll程序集。这里是Default.aspx文件的其他相关页面逻辑（确保按照要求修改连接字符串）：

```
<%@ Page Language="C#" %>
<%@ Import Namespace = "AutoLotConnectedLayer" %>
<%@ Assembly Name ="AutoLotDAL" %>

<!DOCTYPE html>

<script runat="server">
  protected void btnFillData_Click(object sender, EventArgs args)
  {
    InventoryDAL dal = new InventoryDAL();
```

```
        dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
          "Initial Catalog=AutoLot;Integrated Security=True");

        carsGridView.DataSource = dal.GetAllInventoryAsList();
        carsGridView.DataBind();
        dal.CloseConnection();
      }
    </script>
```

在深入探讨这个*.aspx文件幕后的细节之前，让我们做一下测试运行。首先保存*.aspx文件。现在右击*.aspx设计器上的任何位置，选择View in Browser菜单选项，将启动ASP.NET Development Web服务器，它将承载你的页面。

当这个页面运行时，你将最先看到Label和Button控件。然而，当单击按钮控件时，在Web服务器端发生了回传，Web控件发送回它们相应的HTML标签。图32-10显示了单击Fill Grid按钮后的输出结果。

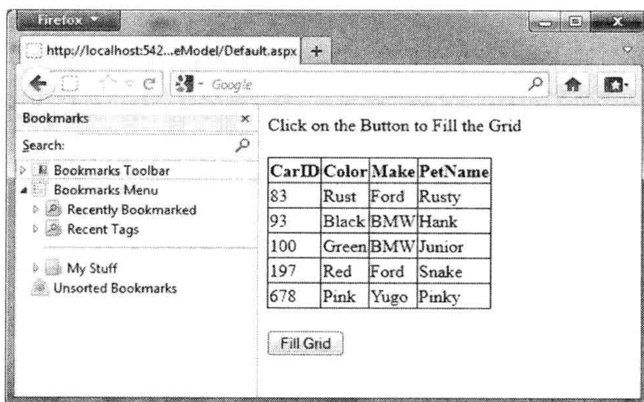


图32-10　ASP.NET提供了声明式数据绑定模型

当前的界面很空。要让它更丰富一些，可在Visual Studio设计器中选择GridView控件，使用上下文菜单（控件左上角的小箭头），选择Auto Format选项（如图32-11所示）。

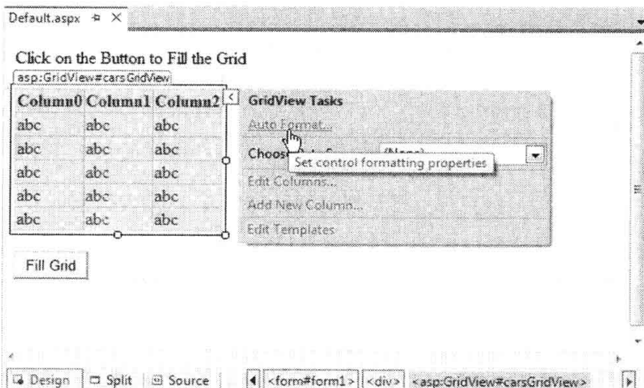


图32-11　配置ASP.NET GridView控件

在弹出的对话框中，选择一个喜欢的模板（我选择“Slate”）。单击OK按钮，查看生成的控件定义，它比之前更丰富一些：

```
<asp:GridView ID="carsGridView" runat="server" BackColor="White"
     BorderColor="#E7E7FF" BorderStyle="None" BorderWidth="1px" CellPadding="3"
     GridLines="Horizontal">
  <AlternatingRowStyle BackColor="#F7F7F7" />
  <FooterStyle BackColor="#B5C7DE" ForeColor="#4A3C8C" />
  <HeaderStyle BackColor="#4A3C8C" Font-Bold="True" ForeColor="#F7F7F7" />
  <PagerStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" HorizontalAlign="Right" />
  <RowStyle BackColor="#E7E7FF" ForeColor="#4A3C8C" />
  <SelectedRowStyle BackColor="#738A9C" Font-Bold="True" ForeColor="#F7F7F7" />
  <SortedAscendingCellStyle BackColor="#F4F4FD" />
  <SortedAscendingHeaderStyle BackColor="#5A4C9D" />
  <SortedDescendingCellStyle BackColor="#D8D8F0" />
  <SortedDescendingHeaderStyle BackColor="#3E3277" />
</asp:GridView>
```

再次查看应用程序并单击按钮，将看到更有趣的界面（如图32-12所示）。

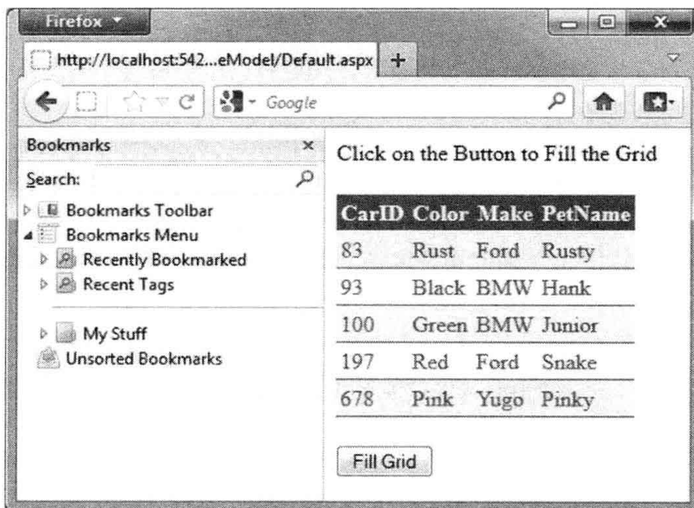


图32-12 显示测试页面的更多内容

很简单吧？当然，正像谚语说的，恶魔藏在细节里[①]，所以让我们更深入地探究这个*.aspx文件的构成。首先研究<%@Page...%>指令。要注意，我们介绍的这些话题也可以直接应用于更受欢迎的代码文件模型中。

32.7.4 ASP.NET指令的作用

通常每个给定的*.aspx文件以一组指令开始。ASP.NET指令总是使用<%@...%>标记表示，并且带有各种特性，以通知ASP.NET运行库如何处理特性。

① 当你要做的事情的最难部分取决于很多细节时，你就可以使用这个谚语。——译者注

每个*.aspx文件至少必须有一个<%@Page%>指令，它用来定义在页面内使用的托管语言（通过language特性）。同时，<%@Page%>指令可以定义相关代码隐藏文件的名字（如果有的话），等等。表32-1给出了一些更有趣的<%@Page%>相关的特性。

表32-1　<%@Page%>指令的部分特性

特　性	作　用
CodePage	指定相关代码隐藏文件的名字
EnableTheming	确定*.aspx页面上的控件是否支持ASP.NET主题
EnableViewState	指定是否在网页请求期间保持视图状态（详细说明见第33章）
Inherits	在代码隐藏页面中定义一个派生出*.aspx文件的类，这个类可以是从System.Web.UI.Page派生出的任何类
MasterPageFile	设置与当前*.aspx页面一同使用的母版页面
Trace	指定是否启用跟踪

除<%@Page%>指令外，给定的*.aspx文件可以指定各种<%@Import%>指令，以显式地声明当前页面需要的命名空间，还可以指定<%@Assembly%>指令以确定站点所使用的外部代码库（通常位于网站的\bin文件夹下）。

在这个例子中，指定正在使用AutoLotAL.dll程序集中AutoLotConnectedLayer命名空间内的类型。可以料到，如果需要使用更多的.NET命名空间，只要指定多个<%@Import%>指令或<%@Assembly%>指令即可。

ASP.NET定义了大量的其他指令，它们会出现在一个*.aspx文件中，并位于<%@Page%>、<%@Import%>和<%@Assembly%>之外，然而，我暂时先不讲这些。在学习剩余章节时，你将看到其他指令的例子。

32.7.5　脚本块

在单文件页面模型下，一个*.aspx文件可能包含在Web服务器端执行的服务器端逻辑脚本。因此所有服务器端代码块都限定在服务器端执行很重要，要使用runat="server"特性。如果不提供runat="server"特性，运行库会假设你编写了一个客户端脚本程序块提交给传出的HTTP响应并且它会抛出异常。下面是合适的服务器端<script>块：

```
<script runat="server">
  protected void btnFillData_Click(object sender, EventArgs args)
  {
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
        "Initial Catalog=AutoLot;Integrated Security=True");
    carsGridView.DataSource = dal.GetAllInventory();
    carsGridView.DataBind();
    dal.CloseConnection();
  }
</script>
```

这个辅助方法的签名你应该相当熟悉。给定的事件处理程序必须匹配由相关的.NET委托定义的模式。委托的是System.EventHandler，它只能调用以System.Object作为第一个参数，System.EventArgs作为第二个参数的方法。

说明 所有ASP.NET Web控件都需要在开放式的声明中添加runat="server"特性。否则，HTML将不能呈现在输出的HTTP响应中。

32.7.6 ASP.NET控件声明

最后介绍一下Button、Label和GridView Web控件的声明。同传统的ASP和原始HTML类似，ASP.NET Web部件必须在<form>元素范围内。但这时，开始标签<form>中包含runat="server"特性，并且所有的控件都用asp:标签前缀进行限定。所有使用该前缀的控件都属于ASP.NET控件库，并且都在.NET基础类库的某个命名空间中包含相应的C#类。例如：

```
<form id="form1" runat="server">
<div>
  <asp:Label ID="lblInfo" runat="server"
    Text="Click on the Button to Fill the Grid">
  </asp:Label>
  <br />
  <br />
  <asp:GridView ID="carsGridView" runat="server">
    ...
  </asp:GridView>
  <br />
  <asp:Button ID="btnFillData" runat="server" Text="Fill Grid" OnClick="btnFillData_Click"/>
</div>
</form>
```

System.Web.dll程序集中的System.Web.UI.WebControls命名空间包含大量ASP.NET Web控件。打开Visual Studio的Object Browser，可以找到DataGrid控件（如图32-13所示）。

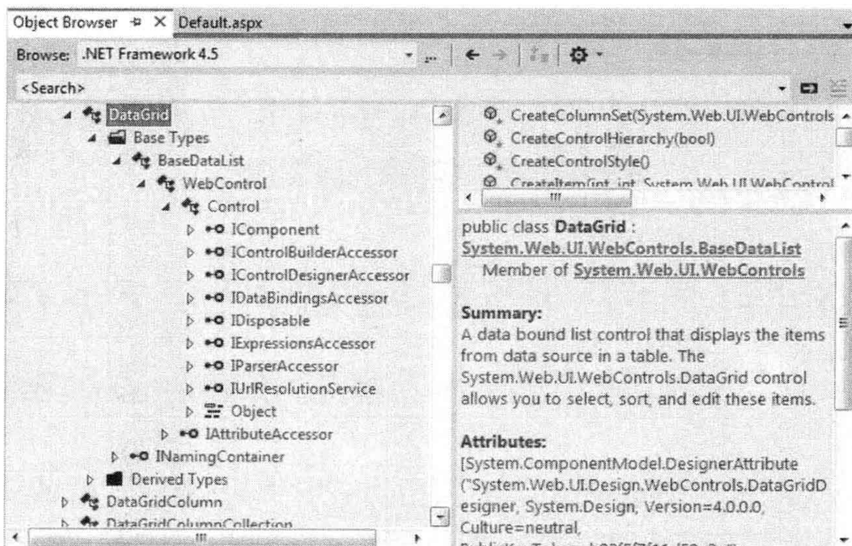


图32-13 所有的ASP.NET控件的声明都与一个.NET类类型相对应

如你所见，ASP.NET Web控件继承链的最顶端为System.Object。所有ASP.NET控件的共同父类为WebControl，它定义了所有公共的UI属性（BackColor、Height等）。Control类在框架中也是很常见的，它定义了与基础结构相关的成员（数据绑定、视图状态等），而不是子类的图形外观。你将在第33章中学习更多关于这些类的知识。

源代码　SinglePageModel页面的源代码位于Chapter 32子目录下。

32.8　使用代码文件构建 ASP.NET Web 页面

尽管单文件代码模型很有帮助，但Visual Studio（在新建Web项目时）默认使用的是代码隐藏技术，它允许你将HTML表现逻辑和服务器端编程代码分别放在两个不同的文件中。该模型尤其适用于页面包含大量代码或多人开发同一个网站的情况。代码隐藏模型还提供了其他一些好处。

- ❑ 代码隐藏页面完全分离了HTML标记和代码，因此可以让设计师操作标记，而让程序员编写C#代码。
- ❑ 对于页面设计师或其他只操作页面标记的人来说，代码是不可见的（你知道，HTML人员对C#代码没什么兴趣）。
- ❑ 代码文件可用于多个*.aspx文件。

不管采用哪种方式，性能上都没有什么不同。实际上，很多ASP.NET Web应用程序都同时使用了这两种方法。为了说明代码隐藏页面模型，我们使用Visual Studio Web Site模板再创建一次先前的示例。打开File→New→Web Site菜单项，选择ASP.NET Empty Web Site模板，如图32-14所示。

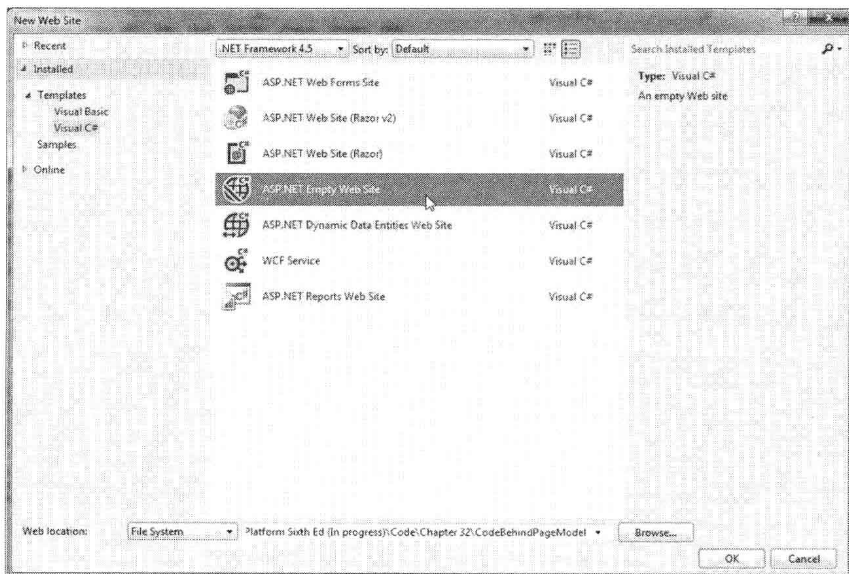


图32-14　Empty Web Site模板

注意在图32-14中，可以选择新站点的位置。如果选择File System，内容文件将放置在一个本地目录中，页面将通过ASP.NET Development Web Server提交。如果选择FTP或者HTTP，站点将驻留在一个由IIS维护的虚拟目录中。对本例来说，你选择哪项是没有差别的，但我建议你选择File System项并且指定一个新的文件夹C:\CodeBehindPageModel。

说明 Empty Web Site项目模板将自动包含一个web.config文件，它与桌面程序中的App.config文件类似。本章稍后将介绍该文件的格式。

现在，使用Website→Add New Item…菜单选项，插入一个新的Web Form项，取名为Default.aspx。你会发现默认情况下，Place code in separate file复选框为选中状态，这正是我们想要的（如图32-15所示）。

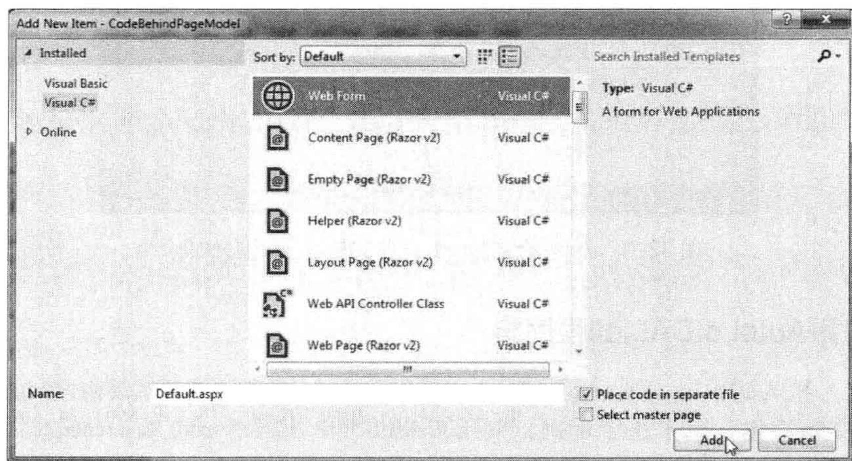


图32-15 添加一个代码分离的Web Form

再次利用设计器创建一个由Label、Button和GridView组成的UI，并使用Properties窗口建立一个你喜欢的UI。如果愿意，可以直接将SingleFilePageModel示例中的ASP.NET控件声明赋值到新的*.aspx文件中。由于是完全相同的标记，在此不会赘述（不过要记得将控件声明粘贴到<form>和</form>标签中。

注意为用于代码文件模型中的<%@ Page%>指令添加如下两个新特性：

```
<%@ Page Language="C#" AutoEventWireup="true"
        CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

CodeFile特性用来指定相关的、包含这个页面的代码逻辑的外部文件。默认情况下，这些代码隐藏文件在*.aspx名字后面加上.cs后缀（本例中是Default.aspx.cs）。如果查看Solution Explorer，通过一个在Web Form图标上的子节点看到这个代码隐藏文件是可见的（如图32-16所示）。

打开代码隐藏文件，你发现派生自System.Web.UI.Page的分部类支持处理Load事件。注意，这个类（_Default）的名字同在<%@Page%>指令内的Inherits特性是一样的：

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {

  }
}
```

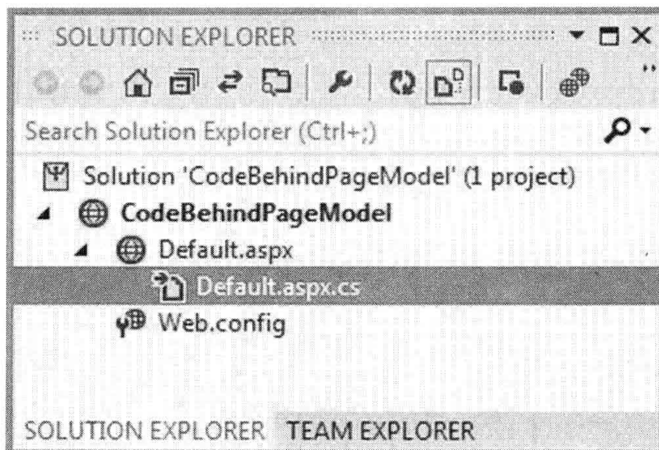


图 32-16　针对给定*.aspx 文件的相关代码隐藏文件

32.8.1　引用AutoLotDAL.dll程序集

之前提过，如果使用Visual Studio创建Web应用程序项目，我们不需要手动构建\bin子文件夹并且手动复制私有程序集。对于这个示例，使用Website菜单项激活Add Reference对话框并且引用AutoLotDAL.dll。然后，就会在Solution Explorer中看到新的\bin文件夹，如图32-17所示。

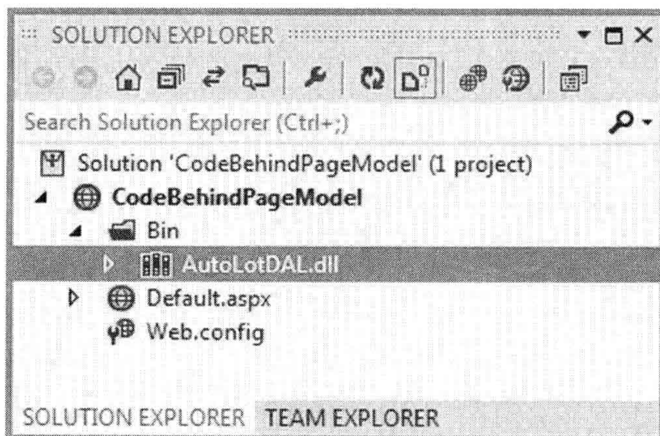


图32-17　Visual Studio Web项目使用特殊的ASP.NET文件夹

32.8.2 更新代码文件

通过双击设计器上的Button为Button类型处理Click事件。像以前一样，Button定义被OnClick特性更新了。然而，服务器端事件处理程序不再放置在*.aspx文件的<script>作用域内，而是作为一种_Default类类型的方法。

为了完成这个例子，在代码隐藏文件内为AutoLotConnectedLayer添加一个using语句，并使用先前的逻辑实现处理程序（同样，请更新连接字符串）：

```
using AutoLotConnectedLayer;

public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
  protected void btnFillData_Click(object sender, EventArgs e)
  {
    InventoryDAL dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
      "Initial Catalog=AutoLot;Integrated Security=True");
    carsGridView.DataSource = dal.GetAllInventoryAsList();
    carsGridView.DataBind();
    dal.CloseConnection();
  }
}
```

现在，按下Ctrl+F5组合键运行网站。ASP.NET Development Web服务器将再次启动，将页面显示在承载的浏览器中。

32.8.3 调试并跟踪ASP.NET页面

当构建ASP.NET Web项目时，你能如愿地使用Visual Studio项目类型中任何种类的调试技术。这样，就能在代码隐藏文件中设置断点（以及在*.aspx文件中嵌入“脚本”块），启动调试会话（默认用F5键），单步调试代码。

然而，为调试ASP.NET Web应用程序，站点必须包括一个正确配置的web.config文件。默认情况下，所有Visual Studio Web项目都会自动具备一个web.config文件。然而，调试支持一开始就是禁用的。如果开始一个调试会话，IDE会提示我们是否修改web.config以启用调试。如果确认启用，在web.config中会按如下所示更新一个<compilation>元素：

```
<compilation debug="true" targetFramework="4.5"/>
```

我们还可以通过将<%@Page%>指令中的Trace特性设置为true来启用*.aspx文件的跟踪支持（还可以通过修改web.config文件来为整个网站启用跟踪）：

```
<%@ Page Language="C#" AutoEventWireup="true"
         CodeFile="Default.aspx.cs" Inherits="_Default" Trace="true" %>
```

一旦这样设置了，已提交的HTML就会包含很多关于先前HTTP请求/响应的细节（服务器变量、会话和应用程序变量，请求/响应，等等）。

可以使用System.Web.UI.Page类型的Trace属性把你自己的跟踪消息插入其中。任何时候想记录自

定义消息日志（从一个脚本块或C#源代码文件）的话，只需调用Trace.Write()方法即可。其第一个参数表示自定义类别的名称，第二个参数指定了跟踪消息。为了演示这一点，使用如下的代码语句更新Button的Click处理程序：

```
protected void btnFillData_Click(object sender, EventArgs e)
{
  Trace.Write("CodeFileTraceInfo!", "Filling the grid!");
...
}
```

如果再一次运行项目并且单击该按钮，将呈现并解释自定义种类和自定义消息。如图32-18所示，注意突出显示的消息，它呈现的是跟踪信息。

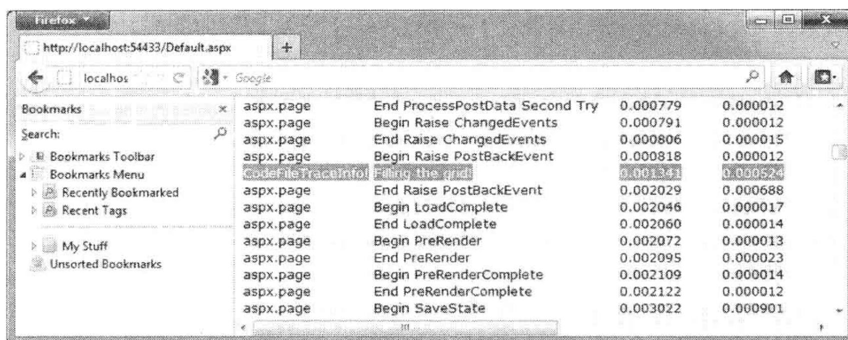


图32-18　记录自定义的跟踪信息

至此，我们已经学习了如何使用单文件和代码文件构建ASP.NET Web页面。本章剩余部分将深入研究ASP.NET Web项目的构成、与HTTP请求／响应交互的方式以及Page派生类的生命周期。在此之前，我需要阐述一下ASP.NET Web Site和ASP.NET Web Application的区别。

源代码　CodeBehindPageModel网站的源代码位于Chapter 32子目录下。

32.9　ASP.NET Web Site 和 ASP.NET Web Application

在构建ASP.NET Web项目时，你需要在两种项目格式之间作出选择，它们是ASP.NET Web Site和ASP.NET Web Application。Visual Studio组织和处理Web应用程序启动文件的方式、所创建的初始项目文件的类型以及对编译的.NET程序集组成部分的控制程度，都取决于你的选择。

当ASP.NET随.NET 1.0发布伊始，我们唯一的选择是Web Application。在这种模型下，可以直接控制编译输出的程序集的名称和位置。

在将旧的.NET 1.1网站向.NET 2.0或更高的版本迁移时，Web Application十分有用。如果希望构建一个包含多个项目的Visual Studio Solution（如包含一个Web Application和3个相关的.NET代码库），它也十分有帮助。要构建ASP.NET Web Application，可以激活File→New Project...菜单项，在Web分类里选择模板（如图32-19所示）。

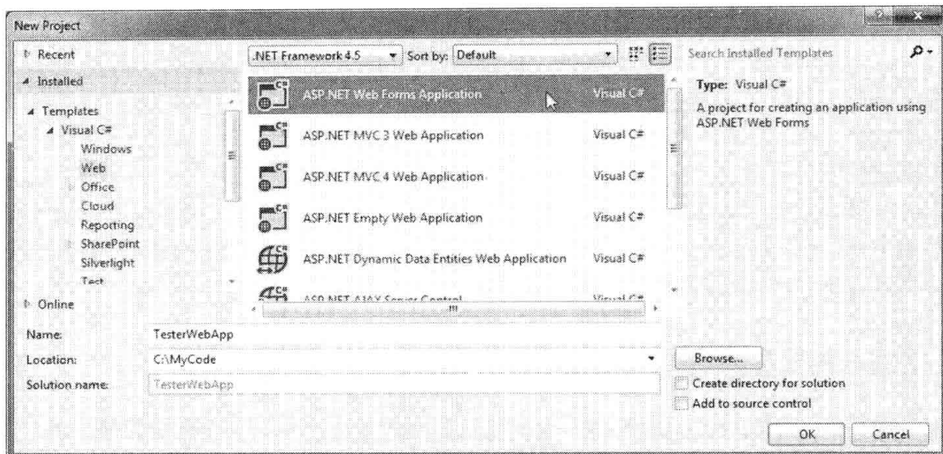


图32-19 Visual Studio Web Application模板

而现在没有必要这么做了。假设你新建了一个ASP.NET Web Application项目，会发现大量的启动文件（在后面的章节中将会介绍），但重要的是要注意每个ASP.NET Web页面都由3个文件组成，*.aspx文件（标记）、*.designer.cs文件（设计器生成的C#代码）和主C#代码文件（事件处理程序和自定义方法等），如图32-20所示。

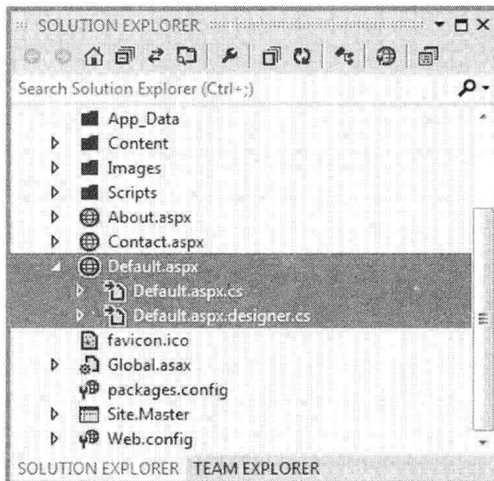


图32-20 在Web Application模型下，每个Web页面由3个文件组成

说明 由于Visual Studio的ASP.NET项目模板生成了大量的启动代码（母版页、内容页、脚本库、页面日志等），本书将使用Blank Web Site模板。但是，学习完本书的ASP.NET内容之后，你应该创建一个ASP.NET Web Site项目，并在第一时间研究这些启动代码。

与之形成鲜明对比的是，Visual Studio中的ASP.NET Web Site项目模板（打开File→New Web Site...菜单选项可以找到）隐藏了*.designer.cs文件，以支持内存分部类。此外，ASP.NET Web Site项目还支持很多特殊名称的文件夹，如App_Code。在该文件夹中，可以放置那些不与Web页面直接映射的C#（或VB）代码文件，运行时编译器将在需要时动态编译。这比单独构建.NET代码库并在新项目中引用这种常见的做法要简单得多。

此外，Web Site项目可以按原样发布，而ASP.NET Web Application需要对网站进行预编译。

我在本书中将使用ASP.NET Web Site项目类型，因为它确实简化了在.NET平台下构建Web应用程序的过程。不过，不管使用哪种方法，都将访问同样的编程模型。

32.10　ASP.NET 网站目录结构

当创建一个新的ASP.NET网站项目时，你的项目可能包括许多特定名字的子目录，其中的每一个对于ASP.NET运行库来说都有一个特定的含义。表32-2列出了这些特殊的子目录。

表32-2　特殊的ASP.NET子目录

子　目　录	作　　用
App_Browsers	浏览器定义文件的文件夹，这些文件用于识别各个浏览器并确定其性能
App_Code	组件或类的源代码的文件夹，这些组件或类是要编译成为应用程序的一部分。当要请求页面时，ASP.NET编译该文件夹里的代码。App_Code文件夹里的代码对应用程序是自动可访问的
App_Data	用于存储Access的*.mdb文件、SQL Express的*.mdf文件、XML文件或其他数据存储的文件夹
App_GlobalResources	用于存储*.resx文件的文件夹，这些文件可以通过编程从应用程序代码访问
App_LocalResources	用于存储*.resx文件的文件夹，这些文件与特定的网页相绑定
App_Themes	包含用来定义ASP.NET网页和控件外观的文件集合的文件夹
App_WebReferences	用于存储代理类、架构和与在应用程序里使用Web服务相关的其他文件的文件夹
Bin	用于存储经过编译的私有程序集（*.dll文件）的文件夹。存放在Bin文件夹里的程序集被应用程序自动引用

如果你想把这些子文件夹添加到你当前的Web应用程序里，可以显式地使用Website→Add ASP.NET Folder菜单项。然而，在许多时候，当你向站点“自然地”添加相关文件时，IDE会自动做这些事情。例如，当你增加一个新的C#文件时，IDE将自动向目录结构添加一个App_Code文件夹，如果该文件夹不存在的话。

32.10.1　引用程序集

虽然Web Site模板生成一个*.sln文件，将*.aspx文件加载到IDE里，但是不再有相关的*.csproj文件。你可能知道，ASP.NET Web Application项目在*.csproj里记录了所有的外部程序集。这个事实带来了明显的问题，在ASP.NET下外部程序集记录到哪儿去了？

当引用一个私有程序集时，Visual Studio将自动在目录结构中创建一个\bin目录以存储一个二进制

的本地副本。当你的代码利用了这些代码库中的类型时，它们会按需自动加载。

如果在GAC中引用一个共享程序集，Visual Studio会自动向当前的Web解决方案中插入一个web.config文件（如果没有这个文件），并且在<assemblies>元素内记录这个外部引用。例如，如果你再打开Web Site→Add Reference菜单项，这次选择一个共享程序集（例如System.Security.dll），你将发现web.config文件被更新了，如下所示：

```
<assemblies>
  <add assembly="System.Security, Version=4.0.0.0,
       Culture=neutral, PublicKeyToken=B03F5F7F11D50A3A"/>
</assemblies>
```

正如所见，描述每个程序集所用的信息，与通过Assembly.Load()方法来动态加载（参见第15章）所需的信息是一样的。

32.10.2 App_Code文件夹的作用

App_Code文件夹用来放置源代码文件，这些文件不直接绑定到一个特定的网页（例如一个代码隐藏文件），但会被编译以备网站使用。App_Code文件夹内的代码将会在需要的时候被自动编译。在这之后，程序集就可以被网站内的任何其他代码访问了。因此，除了能在App_Code文件夹中存储源代码而不是编译后的代码外，它更像Bin文件夹。这种方法的主要优点是，使得为Web应用程序定义自定义类型而不必对它们进行独立编译成为可能。

一个App_Code文件夹能包括多种语言的代码文件。在运行时，适当的编译器会被载入以生成相应的程序集。但是，如果你愿意分割代码，就能定义多个子目录用于保存许多被托管代码文件（*.vb、*.cs等）。

例如，假设你已经在一个网站应用程序的根目录下添加了App_Code文件夹，它包括两个子文件夹（MyCSharpCode和MyVbNetCode），分别包含特定语言的文件。一旦添加了文件夹，你就能够编辑web.config文件，在文件中使用嵌套在<configuration>元素内的<codeSubDirectories>元素指定这些子目录，如下所示：

```
<compilation debug="true" strict="false" explicit="true">
  <codeSubDirectories>
    <add directoryName="MyCSharpCode" />
    <add directoryName="MyVbNetCode" />
  </codeSubDirectories>
</compilation>
```

说明 App_Code目录也将用于包括非语言文件的、但很有用的文件（*.xsd文件、*.wsdl文件等）。

除了Bin和App_Code以外，App_Data和App_Themes文件夹也是我们需要熟悉的另外两个特殊子文件夹，之后几章中我们会详细介绍。同样，如果需要知道更多有关其余ASP.NET子目录的细节，请参考.NET Framework 4.5 SDK文档。

32

32.11 页面类型的继承链

所有.NET网页最终派生自System.Web.UI.Page。像任何基类一样，这个类型提供了一个对所有派

生类型的多态接口。然而，Page类型不是继承层次结构的唯一成员。如果要使用Visual Studio对象浏览器找到System.Web.UI.Page类（在System.Web.dll程序集中），你将发现Page "is-a" TemplateControl，TemplateControl "is-a" Control，Control "is-a" Object（如图32-21所示）。

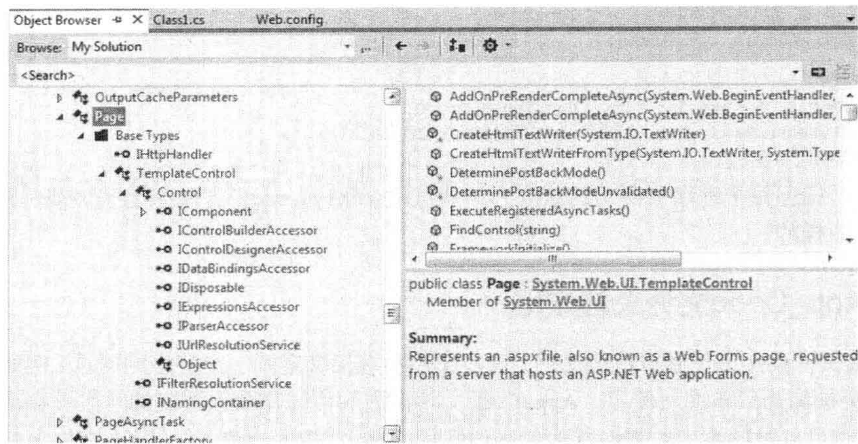


图32-21　页面的继承链

这些基类的每一个都给每个*.aspx文件提供了大量的功能。对于大部分项目，可以利用Page和Control父类中定义的成员。大体上来讲，如果你正在创建自定义Web Form控件或正在与呈现进程交互，你将对从System.Web.UI.TemplateControl类中获得的功能感兴趣。

第一个介绍的父类是Page自身。这里有许多属性，用于与各种基本的Web对象（例如应用程序变量和会话变量、HTTP请求/响应、主题支持等）进行交互。表32-3描述了一些（但不是全部）核心属性。

表32-3　Page类型的属性

属　性	作　用
Application	允许在整个站点内与可以访问的数据进行交互
Cache	允许针对当前站点与高速缓存对象进行交互
ClientTarget	允许指定该页面应当如何根据发出请求的浏览器来呈现自身
IsPostBack	获取一个值，这个值指示正在加载的页面是否响应了客户端回传，或该页面是否是首次被加载和访问
MasterPageFile	为当前页面使用母版页
Request	允许访问当前HTTP请求
Response	允许与发出的HTTP响应进行交互
Server	允许访问HttpServerUtility对象，该对象包含了各种服务器端的帮助功能
Session	允许针对当前调用者与会话数据进行交互
Theme	获取或设置用于当前页面的主题名称
Trace	允许访问TraceContext对象，TraceContext对象允许在调试会话期间记录自定义的信息

32.12 与传入的 HTTP 请求交互

在本章的前文中已经看到，Web应用程序的基本流程开始于客户登录到网页，填写用户信息，然后单击Submit按钮以向给定的网页回传HTML表单数据进行处理。大多数情况下，form语句的开始标签指定一个action特性和一个method特性,这两个特性分别指定各种HTML部件中数据要发送到的Web服务器上的文件，以及发送这个数据（GET或POST）的方法：

```
<form name="defaultPage" id="defaultPage"
      action="http://localhost/Cars/ClassicAspPage.asp" method = "GET">
...
</form>
```

所有的ASP.NET页面都支持System.Web.UI.Page.Request属性，它允许访问一个HttpRequest类类型实例（参见表32-4，其中列出了一些核心成员）。

表32-4 HttpRequest类的成员

成　员	作　用
ApplicationPath	获取ASP.NET应用程序的虚拟应用程序在服务器上的根路径
Browser	提供关于客户端浏览器的功能的信息
Cookies	获取由客户端浏览器发送的cookie的集合
FilePath	指示当前请求的虚拟路径
Form	获取HTTP表单变量的集合
Headers	获取HTTP首部的集合
HttpMethod	指明由客户端使用的HTTP数据传输的方法（GET、POST）
IsSecureConnection	指明HTTP连接是否安全（即HTTPS）
QueryString	获取HTTP查询字符串变量的集合
RawUrl	获取当前请求的原始URL
RequestType	指明由客户端使用的HTTP数据传输的方法（GET、POST）
ServerVariables	获取Web服务器变量的集合
UserHostAddress	获取远程客户端的IP地址
UserHostName	获取远程客户端的DNS名

除了这些属性外，HttpRequest类型还有大量有用的方法，如下所示。

- MapPath()：针对当前请求，将被请求URL中的虚拟路径映射到服务器上的一个物理路径。
- SaveAs()：将当前HTTP请求的细节保存到一个Web服务器上的文件（对于调试是有帮助的）。
- ValidateInput()：如果通过页面指令的Validate特性激活了验证功能，可以调用这个方法，根据预定义的危险输入数据列表来检查所有的用户输入数据（包括cookie数据）。

32.12.1 获得浏览器统计数据

HttpRequest类型的第一个有趣的方面是Browser属性，它提供了对底层HttpBrowserCapabilities对象的访问。HttpBrowserCapabilities则公开了大量的成员，允许你通过编程检查发送传入HTTP请求的浏览器的统计数据。

使用File→New Website菜单选项，并选中使用File System选项创建一个新的ASP.NET站点，命名为FunWithPageMembers。接下来，使用Website→Add New Item菜单选项插入一个新的Web Form文件。

我们的第一个任务是建立一个UI允许用户单击Button Web控件btnGetBrowserStars以浏览发送命令的浏览器的统计数据。这些统计数据将动态生成并附加在一个Label类型（名为lblOutput）上。可以在Web页设计器的任意位置添加两个控件。然后处理Click事件，该处理程序的一种实现如下所示：

```
protected void btnGetBrowserStats_Click(object sender, EventArgs e)
{
  string theInfo = "";
  theInfo += string.Format("<li>Is the client AOL? {0}</li>",
    Request.Browser.AOL);
  theInfo += string.Format("<li>Does the client support ActiveX? {0}</li>",
    Request.Browser.ActiveXControls);
  theInfo += string.Format("<li>Is the client a Beta? {0}</li>",
    Request.Browser.Beta);
  theInfo += string.Format("<li>Does the client support Java Applets? {0}</li>",
    Request.Browser.JavaApplets);
  theInfo += string.Format("<li>Does the client support Cookies? {0}</li>",
    Request.Browser.Cookies);
  theInfo += string.Format("<li>Does the client support VBScript? {0}</li>",
    Request.Browser.VBScript);
  lblOutput.Text = theInfo;
}
```

这是在测试大量的浏览器功能。发现一个浏览器对ActiveX控件、Java applet和客户端VBScript代码的支持是非常有帮助的。如果发起调用的浏览器不支持给定的Web技术，*.aspx页面将可以采取另一个动作。

32.12.2　访问传入的表单数据

HttpRequest类型还有Form和QueryString属性。这两个属性允许使用名称/值对方式检查传入的表单数据。虽然你还是可以利用HttpRequest.Form和HttpRequest.QueryString属性访问Web服务器上由客户端支持的表单数据，但是ASP.NET提供更优雅的面向对象方法。假设ASP.NET支持使用服务器端Web控件，则可以像对待对象一样对待HTML UI元素。因此，不是按如下代码所示在一个文本框里获得值：

```
protected void btnGetFormData_Click(object sender, System.EventArgs e)
{
  // 使用ID txtFirstName为部件获取值
  string firstName = Request.Form("txtFirstName");

  // 在页面中使用这个值
}
```

而只需通过Text属性直接地请求一个服务器端部件：

```
protected void btnGetFormData_Click(object sender, System.EventArgs e)
{
  // 使用ID txtFirstName为部件获取值
  string firstName = txtFirstName.Text;

  // 在页面中使用这个值
}
```

这个方法不仅是纯粹的OO（面向对象）原理，而且不需要你考虑在获得值之前表单数据是如何提交的（GET或POST）。此外，直接使用窗口部件工作更是类型安全的，因为键入的错误是在编译时而不是在运行时发现的。当然，这不是说你永远不需要在ASP.NET中使用Form或QueryString属性，确切地说是，这么做的必要性已经减少了很多。

32.12.3 IsPostBack属性

Page的另一个非常重要的成员是IsPostBack属性。回想一下，“回传”是指页面回传到Web服务器上相同的URL。有了这个定义后，进而理解，如果当前的HTTP请求被一个当前的登录用户发送，则IsPostBack属性将返回true；如果这是用户与页面的第一次交互，则IsPostBack属性将返回false。

通常情况下，仅当用户首次访问一个给定的页面，而你需要执行代码块时，决定当前HTTP请求是否是回传才更有帮助。例如，当用户首次访问*.aspx文件时，你可能希望填充ADO.NET DataSet，同时缓存对象以备下次使用。当调用者返回到页面时，你能避免不必要的对数据库的过度访问（当然，一些页面可能要求DataSet对于每个请求总是更新的，但那是另一个问题）。假设*.aspx文件已经处理了页面的Load事件（本章后面会详细介绍），可以通过编程测试回传条件，如下所示：

```
protected void Page_Load(object sender, EventArgs e)
{
  // 只在用户初次访问页面时填充DataSet
   if (!IsPostBack)
  {
    // 填充DataSet并且缓存它
  }
  // 使用缓存的DataSet
}
```

32.13 与输出 HTTP 响应交互

现在你已经较好地理解了Page类型如何允许与传入HTTP请求交互，下一步是看一看如何与输出HTTP响应交互。在ASP.NET中，Page类的Response属性提供了对HttpResponse类型的实例的访问。这个类型定义了大量的属性，它们允许格式化发送回客户端浏览器的HTTP响应。表32-5列举了一些核心属性。

表32-5 HttpResponse类型的属性

属　性	作　用
Cache	返回网页的高速缓存语义（见第34章）
ContentEncoding	获取或者设置输出流的HTTP字符集
ContentType	获取或者设置输出流的HTTP MIME类型
Cookies	获取将返回到浏览器的HttpCookie集合
Output	对输出HTTP内容主体启用文本输出
OutputStream	对输出HTTP内容主体启用二进制输出
StatusCode	获取或者设置返回客户端的、关于输出的HTTP状态代码
StatusDescription	获取或者设置返回客户端的、关于输出的HTTP状态字符串
SuppressContent	获取或者设置指明HTTP内容不会送回客户端的值

同样，考虑一下表32-6中描述的HttpResponse类型支持的部分方法。

表32-6　HttpResponse类型的方法

方　法	作　用
Clear()	从缓冲流里清除所有首部和内容输出
End()	对客户端发送所有当前经过缓冲的输出，并且在这之后关闭套接字连接
Flush()	对客户端发送所有当前经过缓冲的输出
Redirect()	把客户端重定向到新的URL
Write()	向HTTP输出内容流写入值
WriteFile()	向HTTP内容输出流直接写入一个文件

32.13.1　提交HTML内容

可能HttpResponse类型最著名的一点就是，能直接向HTTP输出流中写内容。通过HttpResponse.Write()方法，可以传送任何HTML标签或文本。HttpResponse.WriteFile()方法使这个功能更进一步，用这个方法你能指定Web服务器上物理文件的名字，这个文件的内容会呈现给输出流（这对于快速提交一个现有的*.htm文件的内容是非常有帮助的）。

举个例子，假设你已经添加了另一个Button类型到当前的*.aspx文件中，这个文件实现了服务器端的Click事件处理程序，如下所示：

```
protected void btnHttpResponse_Click(object sender, EventArgs e)
{
  Response.Write("<b>My name is:</b><br>");
  Response.Write(this.ToString());
  Response.Write("<br><br><b>Here was your last request:</b><br>");
  Response.WriteFile("MyHTMLPage.htm");
}
```

这个辅助函数的作用（你可以假设它被某些服务器端事件处理程序调用）非常简单。唯一需要关注的是，HttpResponse.WriteFile()方法现在正在提交网站根目录里的服务器端*.htm文件的内容。

再次强调，尽管总能够使用这个老方法，用Write()方法呈现HTML标签和内容，但这个方法在ASP.NET下远不及在传统ASP下普遍。理由（再次）应归于出现了服务器端Web控件。这样，如果希望向浏览器呈现一个文本数据块，你的任务就像为Label部件的Text属性赋一个字符串那么简单。

32.13.2　重定向用户

HttpResponse类型的另一个方面就是重定向用户到一个新URL的功能：

```
protected void btnWasteTime_Click(object sender, EventArgs e)
{
  Response.Redirect("http://www.facebook.com");
}
```

如果这个事件处理程序通过客户端回传被调用，用户将自动地被重定向到指定的URL。

说明 HttpResponse.Redirect()方法将总是涉及客户端浏览器的回传。如果你只是希望在同一个虚拟目录里将控件传到一个*.aspx文件，那么HttpServerUtility.Transfer()方法（通过继承的Server属性访问）将更有效。

对System.Web.UI.Page功能的研究到此为止。在下一章中，我们将学习System.Web.UI.Control基类的作用。先来研究Page派生的对象的生命周期和次数。

源代码 FunWithPageMembers文件的源代码位于Chapter 32子目录下。

32.14 ASP.NET 网页的生命周期

每个ASP.NET网页都有一个固定的生命周期。当ASP.NET运行库收到一个给定*.aspx文件的传入请求时，将使用类型的默认构造函数把相关的派生自System.Web.UI.Page的类型分配到内存中。在这之后，框架将自动触发一系列事件。默认情况下，会自动出现Load事件，我们可以直接增加自己的自定义代码：

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    Response.Write("Load event fired!");
  }
}
```

除了Load事件，给定的Page能截取表32-7中任何核心事件，这些事件按照被截取的次序列在表32-7中（.NET Framework 4.5 SDK文档详细介绍了页面生命周期内可能触发的所有事件）。

表32-7 Page类型的部分事件

事 件	作 用
PreInit	框架使用该事件来分配Web控件，应用主题，确立母版页，并设置用户个性化配置。你可以截取该事件来定制进程
Init	框架使用该事件通过回传或查看状态数据把Web控件的属性设置为它们先前的值
Load	当该事件触发时，页面与其控件被完全初始化，并且它们先前的值得以恢复。此刻，与各个Web窗口部件进行交互都是安全的
引发回传的事件	当然不存在具有这个名字的事件。该“事件”仅仅是指任何导致浏览器对Web服务器发出回传的事件（例如单击按钮）
PreRender	所有控件数据绑定和UI配置已经发生，并且控件已经准备好用于将它们的数据呈现到将要发出的HTTP响应里
Unload	页面与其控件已经完成了呈现过程，并且页面对象将被销毁。此刻，与输出HTTP响应进行交互会出现运行时错误。然而你可以捕捉该事件来执行任何页面层的清除（关闭文件或数据库连接，执行任何形式的记录活动，处置对象，等等）

如果需要处理除了Load之外的事件，你会发现要实现这个是没有IDE支持的！我们必须在代码文件中手动编写方法，名字为“Page_事件名”。例如，我们是这样处理Unload事件的：

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    Response.Write("Load event fired!");
  }

  protected void Page_Unload(object sender, EventArgs e)
  {
    // 不能再向HTTP响应输出数据，因为我们会写入本地文件
    System.IO.File.WriteAllText(@"C:\MyLog.txt", "Page unloading!");
  }
}
```

说明　Page类型的每一个事件都和System.EventHandler委托一起使用。因此，处理这些事件的子程序总是接受Object作为第一个参数，接受EventArgs作为第二个参数。

32.14.1　AutoEventWireUp特性的作用

当你希望为页面处理事件时，需要使用一个适当的事件处理程序更新<script>块或者代码隐藏文件。然而，如果检查<%@Page%>指令，就会注意到AutoEventWireUp特性在默认情况下设置为true：

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

有了这个默认的行为，如果我们输入正确命名的方法，页面级别的事件处理程序会被自动处理。然而，如果通过将这个特性设置为false来禁用AutoPageWireUp的话：

```
<%@ Page Language="C#" AutoEventWireup="false"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

页面级别的事件不再会被捕获。正如其名，这个特性（如果启用）会在自动生成的分部类（本章前面提到过）中生成必要的事件绑定。即使禁用AutoEventWireUp，我们仍然可以通过使用C#的事件处理逻辑来处理页面级别的事件，例如：

```
public _Default()
{
  // 显式挂接Load和Unload事件
  this.Load += Page_Load;
  this.Unload += Page_Unload;
}
```

在大多数情况下，我们只需要保持AutoEventWireUp为可用状态。

32.14.2　Error事件

另一个可能在页面生命周期中发生的事件是Error。如果一个派生自Page的类型上的方法触发了一个不能被明确处理的异常，这个事件将被触发。假设你已经为一个页面上给定Button处理了Click

事件，并且在事件处理程序（我称之为btnGetFile_Click）中，你试图向HTTP响应写入一个本地文件的内容。

同样假设你无法通过标准结构异常处理进行该文件的存在性测试。如果你已经加载了默认构造函数中页面的Error事件，你还有最后一个机会在用户发现难看的错误前处理问题。思考下面的代码：

```
public partial class _Default : System.Web.UI.Page
{
  void Page_Error(object sender, EventArgs e)
  {
    Response.Clear();
    Response.Write("I am sorry...I can't find a required file.<br>");
    Response.Write(string.Format("The error was: <b>{0}</b>",
        Server.GetLastError().Message));
    Server.ClearError();
  }

  protected void Page_Load(object sender, EventArgs e)
  {
    Response.Write("Load event fired!");
  }

  protected void Page_Unload(object sender, EventArgs e)
  {
    // 不能再向HTTP响应输出数据，因为我们会写入本地文件
    System.IO.File.WriteAllText(@"C:\MyLog.txt", "Page unloading!");
  }

  protected void btnPostback_Click(object sender, EventArgs e)
  {
    // 没有任何东西，只是为了进行页面回发
  }
  protected void btnTriggerError_Click(object sender, EventArgs e)
  {
    System.IO.File.ReadAllText(@"C:\IDontExist.txt");
  }
}
```

注意，Error事件处理程序开始清除HTTP响应中的任何当前内容，并且发送一个普通的错误消息。如果你希望访问指定的System.Exception对象，可以使用由继承的Server属性公开的HttpServerUtility.GetLastError()方法：

```
Exception e = Server.GetLastError();
```

最后注意，在退出这个普通的错误处理程序之前，要通过Server属性显式地调用HttpServer Utility.ClearError()方法。这是必需的，因为它通知运行库你已经处理了遇到的问题并且不需要再处理了。如果忘记了这么做，呈现在终端用户面前的就是一个运行时错误的页面。

至此，你应该非常了解ASP.NET的Page类型的构成了。现在，你已经具备了这样一个基础，可以把注意力放到ASP.NET Web控件、主题和母版页（后面的章节将介绍它们）的作用上了。在本章结束前，介绍一下web.config文件的作用。

源代码　PageLifeCycle文件的源代码位于Chapter 32子目录下。

32.15 web.config文件的作用

默认情况下，所有使用Visual Studio创建的C#ASP.NETWeb应用程序都会自动具有web.config文件。然而，如果你希望为网站手动插入web.config文件的话（例如，如果我们使用单页面模型并且还没有创建Web解决方案），就可以使用Website→Add New Item菜单项来完成。不管怎么样，在web.config文件中我们都可以增加设置来控制Web应用程序在运行时的工作方式。

回忆一下，在研究.NET程序集的时候（在第14章），我们知道客户端应用程序可以利用基于XML的配置文件来指导CLR处理绑定请求、程序集探查以及其他运行时细节。对于ASP.NET Web应用程序来说也是一样，只是Web相关的配置文件总是叫web.config（和*.exe配置文件不同，它会根据相关客户端可执行文件来命名）。

web.config文件的整体结构相当烦琐。表32-8列出了可以在web.config文件中找到的一些更有趣的子元素。

表32-8 web.config文件的部分元素

元素	作用
<appSettings>	这个元素用于构建自定义名称/值对，它可以使用ConfigurationManager类型以编程方式读入内存使用
<authentication>	这个安全相关的元素用于定义Web应用程序的验证模式
<authorization>	这是另外一个安全相关的元素，它用于定义哪些用户可以访问Web服务器上的哪些资源
<connectionStrings>	这个元素用于保存网站中外部的连接字符串
<customErrors>	这个元素用于告诉运行库如何显示在Web应用程序工作时候发生的异常
<globalization>	这个元素用于为Web应用程序配置全球化设置
<namespaces>	如果我们的应用程序使用新的aspnet_compiler.exe命令行工具预编译的话，这个元素列举了所有需要包含的命名空间
<sessionState>	这个元素用于控制会话状态数据如何以及在何地被.NET运行库存储
<trace>	这个元素用于为Web应用程序启用（或禁用）跟踪支持

除了表32-8列出的其他子元素以外，web.config文件可能会包含其他子元素。表中大部分都是安全相关的，其余的项只是在创建诸如自定义HTTP头或自定义HTTP模块等高级ASP.NET应用的时候才有用（这里不会介绍这些主题）。

ASP.NET网站管理工具

尽管我们完全可以直接使用Visual Studio来修改web.config文件的内容，但ASP.NET Web项目可以使用方便的编辑器以图形方式编辑web.config文件的许多元素和特性。只需要激活Website→ASP.NET Configuration菜单项，就可以启用这个工具。

单击页面顶部的标签页，就可以发现这个工具的大部分功能都是用于为我们的网站创建安全设置的。不过，这个工具还能为<appSettings>元素增加配置，定义调试和跟踪设置以及创建默认错误页面。

必要时，你可以在实战中使用这个工具，然而，要知道这个工具不能帮我们向web.config文件增加所有可能的设置。这个时候你可以选择喜欢的文本编辑器来手动更新这个文件。

32.16 小结

构建Web应用程序需要和过去构建传统的桌面应用程序不同的思路。在本章中，我们首先快速轻松地学习了一些核心的Web主题，包括HTML、HTTP、客户端脚本的作用以及使用传统ASP的服务器端脚本。本章大部分内容都在介绍ASP.NET页面的架构。我们看到了，项目中每一个*.aspx文件都关联了一个从`System.Web.UI.Page`派生的类。使用这种面向对象的方式，我们可以构建重用性更强、更面向对象的系统。

在研究了页面继承链的一些核心功能之后，本章讨论了页面最后如何完整编译为有效的.NET程序集。我们最后研究了web.config文件，并且回顾了ASP.NET网站管理工具。

第33章

ASP.NET Web控件、母版页和主题

第32章讨论了ASP.NET Web页面的一般构成和Page类的行为，而本章会深入介绍构成页面用户界面的Web控件的细节。在研究了ASP.NET Web控件的总体特性之后，我们会理解如何使用包括验证控件和各种数据绑定技术在内的几个UI元素。

本章大部分内容会研究*母版页*的作用，并且演示它们如何提供简单的形式来建立会在网站多个页面公用的UI骨架。与母版页密切相关的话题是站点导航控件（及相关的*.sitemap文件），它可以通过服务器端的XML文件为多页面站点定义导航结构。

最后，你将学习ASP.NET主题的作用。从概念上讲，主题与CSS的目的相同。但ASP.NET主题是在Web服务器端使用（而不是客户端浏览器），并因此可以访问服务器端资源。

33.1 Web 控件的本质

可能ASP.NET的主要优点是使用定义在System.Web.UI.WebControls命名空间里的类型装配页面UI的能力。你已经看到，这些控件（以*服务器控件*、*Web控件*或者*Web窗体控件*命名）是非常有帮助的，它们为发出请求的浏览器自动生成必需的HTML，并公开一套可在Web服务器上处理的事件。而且，因为每一个ASP.NET控件在System.Web.UI.WebControls命名空间里都有一个相应的类，它能够以OO方式操作。

可见，当使用Visual StudioProperties窗口为Web控件配置属性时，你的编辑工作会以一系列的名称/值对记录在*.aspx文件中一个给定元素的公开控件标记中。因此，如果为给定*.aspx文件的设计器添加一个新的TextBox，并且改变ID、BorderStyle、BorderWidth、BackColor和Text属性，公开的<asp:TextBox>标签就会被修改成这样（注意Text值成了TextBox的内部文本）：

```
<asp:TextBox ID="txtNameTextBox" runat="server" BackColor="#C0FFC0"
    BorderStyle="Dotted" BorderWidth="3px">Enter Your Name</asp:TextBox>
```

由于Web控件的声明最后变成来自System.Web.UI.WebControls命名空间的成员变量（通过动态编译周期，第32章介绍过），你就能与在服务器端<script>块内这个类型的成员或通过更常用的页面代码隐藏文件进行交互。例如，如果向*.aspx文件添加一个新的Button按钮，就可以处理Click事件，并且可以编写服务器端处理程序改变TextBox的背景颜色，如下所示：

```
partial class _Default : System.Web.UI.Page
{
  protected void btnChangeTextBoxColor_Click(object sender, EventArgs e)
```

```
    {
      // 改变文本框对象的颜色
      this.txtNameTextBox.BackColor = System.Drawing.Color.DarkBlue;
    }
  }
```

所有的ASP.NET Web控件最终派生自一个公共的名为System.Web.UI.WebControls.WebControl的基类。WebControl派生自System.Web.UI.Control（它又继承自System.Object）。Control和WebControl都定义了大量的所有服务器端控件公共的属性。在学习继承功能之前，正式说明一下处理服务器端事件意味着什么。

33.1.1 服务器端事件处理

考虑到当前的万维网状态，我们不可能避免浏览器/Web服务器的交互。无论何时这两个实体进行通信，总是有一个底层的、无状态的HTTP请求-响应循环。尽管ASP.NET服务器控件做了大量工作以避免用户处理原始HTTP协议的细节，但一定要牢记一点，把Web当做事件驱动的实体来看待仅仅是.NET平台提供的一个华丽的表象，而且它与基于Windows的桌面GUI框架（如WPF）并不完全一样。

因此，虽然WPF的System.Windows.Controls和ASP.NET的System.Web.UI.WebControls命名空间都使用同样简单的名字（Button、TextBox、Label等）定义了类，但它们提供的属性、方法和事件集不同。例如，当用户在一个Web窗体的Button控件上移动光标时，绝不可能处理一个服务器端MouseMove事件。

底线是给定的ASP.NET Web控件将公开一个有限的事件集，所有这些事件最终都将产生到Web服务器的回传。任何必需的客户端事件处理都需要你对客户端JavaScript/VBScript脚本代码有所了解，这些代码将由发出请求的浏览器的脚本引擎处理。因为ASP.NET主要是服务器技术，所以在本书中我们不讨论编写客户端脚本方面的内容。

说明 使用Visual Studio为某个Web控件处理事件和为Windows GUI控件处理时做法相似。只需要从设计器选择控件并且在Properties窗口中单击“闪电”图标。

33.1.2 AutoPostBack属性

同样值得指出的是，ASP.NET的许多Web控件都支持名为AutoPostBack的属性（最主要的是CheckBox、RadioButton和TextBox控件，还有派生自抽象的ListControl类型的部件）。默认情况下，这个属性设置为false，它使服务器端的事件不能自动记录（即使你确实已经在代码隐藏文件中装载了事件）。在许多情况下，这正是你需要的行为，因为诸如复选框的UI元素通常不需要回发功能。也就是说，你并不需要在用户选中或取消选中复选框后立即向服务器回发，因为页面对象可以在Button的Click事件处理程序中获取部件的状态。

然而，如果你希望这些部件中的任何一个回传到服务器端事件处理程序，只需设置AutoPostBack的值为true。如果你希望一个部件的状态自动填充同一页面上另一个部件的值，这个技术可能是有帮助的。例如，创建一个包含TextBox（名为txtAutoPostback）和ListBox控件（名为lstTextBoxData）的网页。下面是相关标记：

```
<form id="form1" runat="server">
  <asp:TextBox ID="txtAutoPostback" runat="server"></asp:TextBox>
  <br/>
  <asp:ListBox ID="lstTextBoxData" runat="server"></asp:ListBox>
</form>
```

现在，处理TextBox的TextChanged事件，并在服务器端事件处理程序中使用TextBox中的当前值填充ListBox，如下所示：

```
partial class _Default : System.Web.UI.Page
{
  protected void txtAutoPostback_TextChanged(object sender, EventArgs e)
  {
    lstTextBoxData.Items.Add(txtAutoPostback.Text);
  }
}
```

如果运行这个应用程序，你将发现，同在TextBox中输入的一样，什么都没有发生。此外，如果你在TextBox中输入并转到下一个控件，同样什么都不会发生。原因是TextBox的AutoPostBack属性默认设置为了false。然而，如果设置这个属性为true，如下所示：

```
<asp:TextBox ID="txtAutoPostback"runat="server"
             AutoPostBack="true" ... >
</asp:TextBox>
```

你将发现，当离开TextBox（或按Enter键）时，ListBox会被TextBox中的当前值自动填充。要记住，除了从一个部件获取值填充到另一个部件中，通常不需要改变部件的AutoPostBack属性的状态（有时可完全用客户端脚本完成，而不需要与服务器交互）。

33.2　Control 和 WebControl 基类

System.Web.UI.Control基类定义了各种属性、方法和允许与Web控件的核心（通常是非GUI的）方面进行交互的事件。表33-1列出了一些成员，但不是全部。

表33-1　System.Web.UI.Control的部分成员

成　员	作　用
Controls	该属性获取一个ControlCollection对象，该对象表示当前控件内的各个子控件
DataBind()	该方法将一个数据源绑定到被调用的服务器控件及其所有子控件
EnableTheming	该属性确定控件是否支持主题功能（默认为true）
HasControls()	该方法确定服务器控件是否含有任何子控件
ID	该属性获取或者设置分配给服务器控件的程序标识符
Page	该属性获取对含有服务器控件的Page实例的引用
Parent	该属性获取对网页控件层次中服务器控件的父控件的引用
SkinID	该属性获取或者设置“皮肤”以应用于控件。它通过服务器端资源迅速建立一个控件的总体外观
Visible	该属性获取或者设置一个值，指明服务器控件是否以UI元素的形式呈现在网页上

33.2.1 枚举所包含的控件

我们将要考查的System.Web.UI.Control的第一个方面是，所有Web控件（包括Page自身）继承一个自定义控件集合（通过Controls属性访问）。与在一个WindowsForms应用程序中非常类似，Controls属性提供了访问WebControl派生类型的强类型集合的方法。同所有.NET集合一样，你可以在运行时动态添加、插入和删除项。

尽管直接将Web控件添加到一个派生自Page的类型在技术上是可行的，但使用Panel控件要容易得多(更健壮)。Panel类描述了一个部件的容器，它对于终端用户可以可见，也可以不可见(取决于Visible和BorderStype属性的值)。

为举例说明，创建一个新的名为DynamicCtrls的站点并向该项目中添加一个新Web Form。使用Visual Studio页面设计器，添加一个Panel控件（名为myPanel），它包含TextBox、Button和HyperLink部件，名字随你选择（注意，设计器需要你在Panel类型的UI中拖动内部项）。接着将Label部件放在Panel（名为lblControlInfo）的作用域之外以保存呈现的输出。下面是可能的HTML描述：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Dynamic Control Test</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
    <hr />
    <h1>Dynamic Controls</h1>
    <asp:Label ID="lblTextBoxText" runat="server"></asp:Label>
    <hr />
    </div>

    <!--Panel有3个包含控件-->
    <asp:Panel ID="myPanel" runat="server" Width="200px"
         BorderColor="Black" BorderStyle="Solid" >
      <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox><br/>
      <asp:Button ID="Button1" runat="server" Text="Button"/><br/>
      <asp:HyperLink ID="HyperLink1" runat="server">HyperLink
      </asp:HyperLink>
    </asp:Panel>
  <br />
  <br />
  <asp:Label ID="lblControlInfo" runat="server"></asp:Label>
</form>
</body>
</html>
```

使用这个标记，页面设计器看起来如图33-1所示。

假设在Page_Load()事件中，你希望获取一个包括在Panel内的所有控件的列表，并且把结果赋值给Label控件（名为lblControlInfo）。考虑下面的C#代码：

```
public partial class _Default : System.Web.UI.Page
{
  private void ListControlsInPanel()
  {
    string theInfo = "";
    theInfo = string.Format("<b>Does the panel have controls? {0} </b><br/>",
```

```
      myPanel.HasControls());

    // 获取面板中的所有控件
    foreach (Control c in myPanel.Controls)
    {
      if (!object.ReferenceEquals(c.GetType(),
        typeof(System.Web.UI.LiteralControl)))
      {
        theInfo += "***************************<br/>";
        theInfo += string.Format("Control Name? {0} <br/>", c.ToString());
        theInfo += string.Format("ID? {0} <br>", c.ID);
        theInfo += string.Format("Control Visible? {0} <br/>", c.Visible);
        theInfo += string.Format("ViewState? {0} <br/>", c.EnableViewState);
      }
    }
    lblControlInfo.Text = theInfo;
  }

  protected void Page_Load(object sender, System.EventArgs e)
  {
    ListControlsInPanel();
  }
}
```

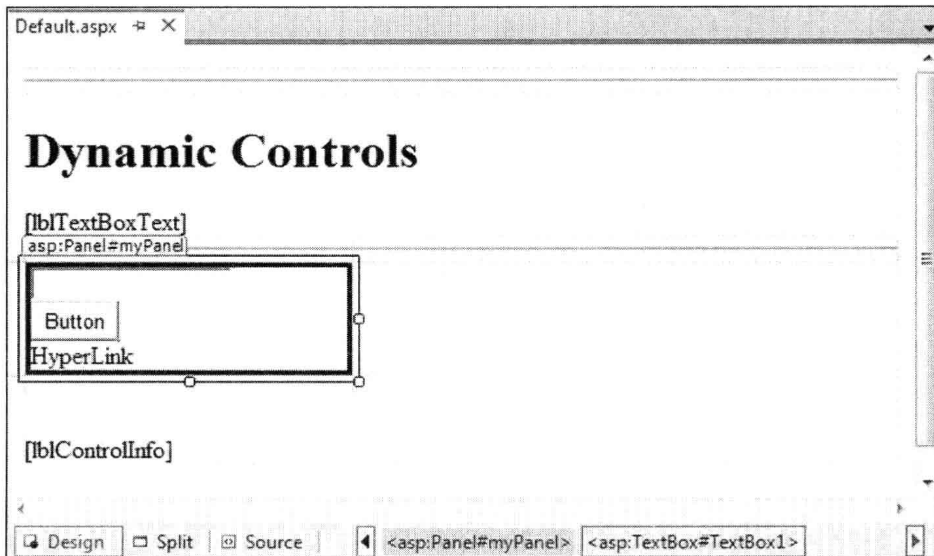


图33-1　Dynamic Controls网页的UI

这里迭代了Panel上的每个WebControl并且检查了当前的类型是否是System.Web.UI.LiteralControl。这个类型用于描述字面量HTML标签和内容（例如
、文本字面量等）。如果不做这个全面检查，你可能会对在Panel（假设*.aspx声明预先可见）的作用域内发现更多类型感到惊讶。假设这个类型不是字面量HTML内容，需要用户打印出关于部件的各类统计。图33-2显示了输出结果。

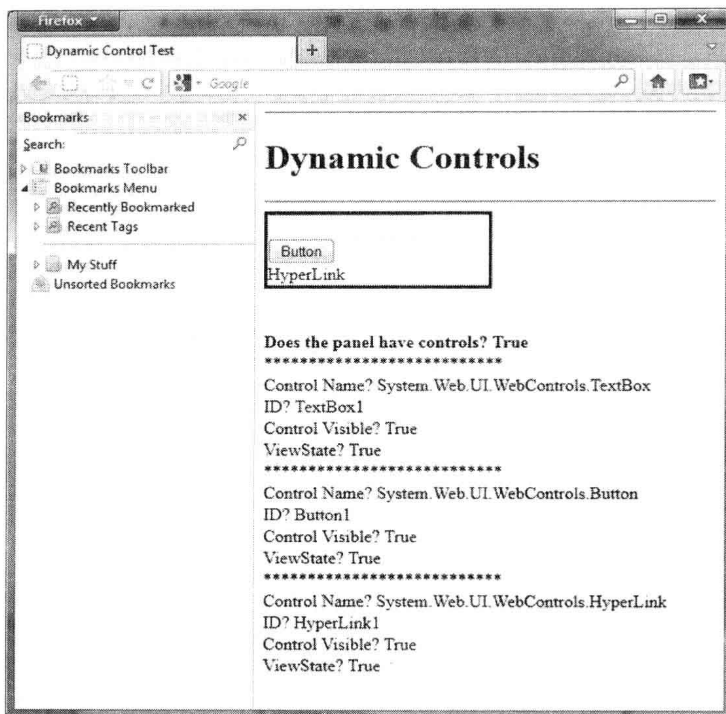


图33-2 在运行时枚举控件

33.2.2 动态添加和删除控件

现在假如希望在运行时改变Panel的内容，该怎么处理呢？更新当前的页面以支持另外一个Button（名为btnAddWidgets），这个按钮动态地向Panel上添加3个新的TextBox类型，另一个Button（名为btnClearPanel）清除所有控件的Panel部件。每个部件的Click事件处理程序如下所示：

```
protected void btnClearPanel_Click(object sender, System.EventArgs e)
{
  // 清除面板中的所有内容，然后重新列出所有项
  myPanel.Controls.Clear();
  ListControlsInPanel();
}

protected void btnAddWidgets_Click(object sender, System.EventArgs e)
{
  for (int i = 0; i < 3; i++)
  {
    // 分配一个名字，这样随后我们就能够使用传入的表单数据获得文本值
    TextBox t = new TextBox();
    t.ID = string.Format("newTextBox{0}", i);
    myPanel.Controls.Add(t);
    ListControlsInPanel();
  }
}
```

注意，为每个TextBox（例如，newTextBox0、newTextBox1等）分配一个唯一的ID。如果运行方面，应该能向Panel控件添加新项并且清除所有内容的Panel。

33.2.3　与动态创建的控件交互

现在，如果想获取这些动态生成的TextBox中的值，可以用多种方法来实现。首先，用另外的Button（名为btnGetTextData）和最后的Label控件（名为LblTextBoxData）更新UI，并且处理Button的Click事件。

为了访问动态生成的文本框中的数据，可以有几种方法。一种方法是循环遍历包含在传入HTML表单数据内的每一项（通过HttpRequest.Form访问），并且拼接文本信息到一个本地作用域System.String。遍历整个集合之后，立即将这个字符串分配到新Label控件的Text属性，如下所示：

```
protected void btnGetTextData_Click(object sender, System.EventArgs e)
{
  string textBoxValues = "";
  for (int i = 0; i < Request.Form.Count; i++)
  {
    textBoxValues += string.Format("<li>{0}</li><br/>", Request.Form[i]);
  }
  lblTextBoxData.Text = textBoxValues;
}
```

当运行这个应用程序时，你将发现你能查看每一个文本框的内容，包括一个相当长的（难读的）字符串。这个字符串包括页面上每个窗口部件的视图状态。在第34章中我们将学习视图状态的作用。

为了使输出结果变得整洁，可以提取各个命名项（newTextBox0、newTextBox1和newTextBox2）的文本数据。考虑如下的更新：

```
protected void btnGetTextData_Click(object sender, System.EventArgs e)
{
  // 通过名称获取各文本框
  string lableData = string.Format("<li>{0}</li><br/>",
    Request.Form.Get("newTextBox0"));
  lableData += string.Format("<li>{0}</li><br/>",
    Request.Form.Get("newTextBox1"));
  lableData += string.Format("<li>{0}</li><br/>",
    Request.Form.Get("newTextBox2"));
  lblTextBoxData.Text = lableData;
}
```

此外，注意，一旦执行了请求，文本框就要消失，这是由于HTTP无状态的特征。如果你希望动态地保持这些被创建于回传期间的TextBox，就需要使用ASP.NET的状态编程技术（也将在第34章中介绍）。

源代码　DynamicCtrls网站的源代码位于Chapter 33子目录下。

33.2.4　WebControl基类的功能

如你所知，Control类型提供了大量与非GUI相关的行为（如控件集合、自动回传支持等）。另一

方面，WebControl基类提供了一个到所有Web部件的图形多态接口，如表33-2所示。

表33-2 WebControl基类的部分属性

属　性	作　用
BackColor	获取或者设置Web控件的背景颜色
BorderColor	获取或者设置Web控件的边框颜色
BorderStyle	获取或者设置Web控件的边框样式
BorderWidth	获取或者设置Web控件的边框宽度
Enabled	获取或者设置用来指示Web控件是否启用的值
CssClass	允许把一个类分配给一个Web部件，这个类是在CSS内定义的
Font	获取Web控件的字体信息
ForeColor	获取或者设置Web控件的前景色（通常情况下为文本的颜色）
Height和Width	获取或者设置Web控件的高度和宽度
TabIndex	获取或者设置Web控件的选项卡索引
ToolTip	获取或者设置当光标位于Web控件之上时，用以显示说明该Web控件的工具提示

我认为，这些属性中的大部分是无需解释的，所以不要钻研所有这些属性的使用。做一下调整：在实际应用中学习大量的ASP.NET Web窗体控件。

33.3 ASP.NET Web 控件的类别

ASP.NET Web控件库可以细分为几大类，可以在Visual Studio Toolbox中查看这几大类（需要打开*.aspx页面的设计视图），如图33-3所示。

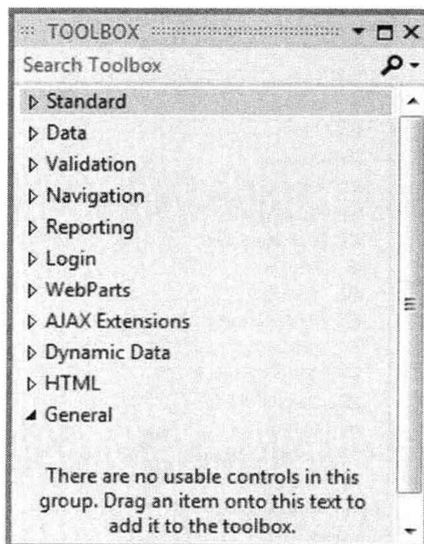


图33-3 ASP.NET Web控件的种类

在Toolbox的Standard节点下，包含最常用的控件，如Button、Label、TextBox和ListBox。除了这些简单的UI元素，Standard节点还列出了更加复杂的Web控件，如Calendar、Wizard和AdRotator（如图33-4所示）。

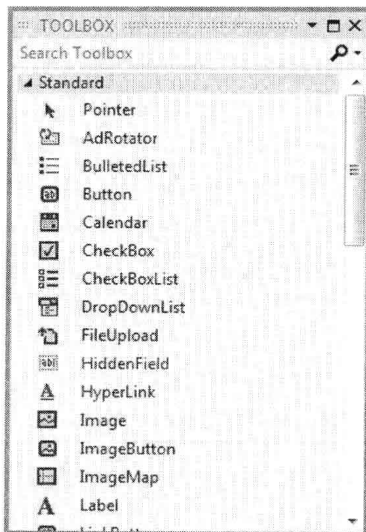


图33-4　标准的ASP.NET Web控件

Data节点下为一组用于数据绑定操作的控件，包括最新的ASP.NET Chart控件，它将图形图表数据（饼图、折线图）作为数据绑定操作的结果进行呈现（如图33-5所示）。

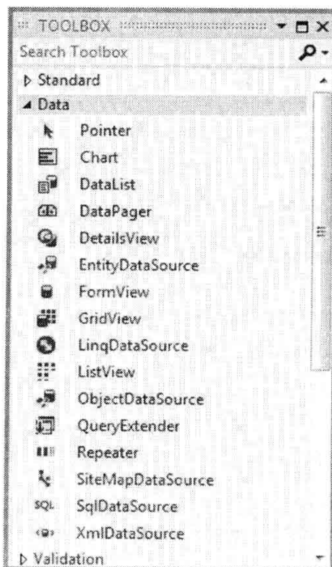


图33-5　数据相关的ASP.NET Web控件

位于Validation节点下的ASP.NET验证控件可以配置客户端JavaScript块，以验证输入字段中的数据是否有效。如果发生验证错误，用户将看到错误消息，并且在错误更正之前不会对Web服务器进行回发。

Toolbox的Navigation节点下是少量与*.sitemap文件协同工作的控件（Menu、SiteMapPath和TreeView）。本章前面简要介绍过，这些导航控件允许使用XML来描述多页面站点的结构。

最神奇的ASP.NET Web控件位于Login节点下（如图33-6所示）。

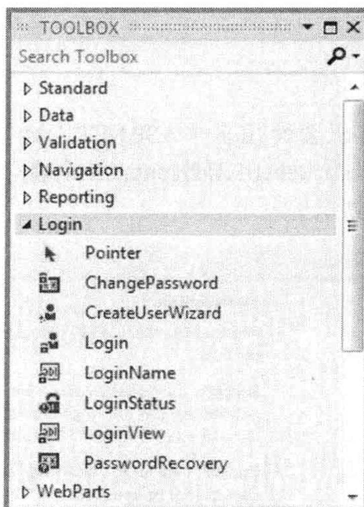


图33-6 完全相关的ASP.NET Web控件

这些控件从根本上简化了如何将基本的安全特性（密码恢复、登录界面等）添加到Web应用程序中。这些控件实际上非常强大，如果你还没有指定安全数据库，它们甚至可以动态创建一个专用的数据库来存储凭证（保存在网站的App_Data文件夹下）。

说明 Visual Studio Toolbox中其他类别的Web控件（如WebParts、AJAX Extensions和Dynamic Data）可以满足更特殊的编程需要，这里不作介绍。

33.3.1 关于System.Web.UI.HtmlControls的简短说明

实际上，ASP.NET中有两个明显不同的Web控件工具包。除了ASP.NET Web控件（在System.Web.UI.WebControls命名空间中）外，基础类库也提供System.Web.UI.HtmlControls控件库。

HTML控件是一个类型的集合，它允许在Web Form页面上使用传统的HTML控件。然而，与原始的HTML标签不同，HTML控件是OO实体，能够被配置并在服务器上运行，因而支持服务器端事件处理。与ASP.NET Web控件不同，HTML控件实际上是相当简单的，并且除了标准HTML标签（HtmlButton、HtmlInputControl、HtmlTable等）之外几乎没什么功能。

如果我们的团队对于构建HTML UI和.NET开发分工明确的话，HTML控件就很有用。HTML的构建者可以通过他们熟悉的标记标签来使用他们的Web编辑器并且把HTML文件传给开发组。然后，开发人员就可以配置这些HTML控件来作为服务器控件运行（通过在Visual Studio中右击HTML组件）。这就允许开发人员来处理服务器端事件以及以编程方式使用HTML控件。

HTML控件提供了一个模仿标准HTML特性的公共接口。例如，为了获得输入区域的信息，你利用`Value`属性，而不是以Web控件为中心的`Text`属性。由于HTML控件不像ASP.NET Web控件那样功能丰富，后文中将不会进一步介绍。

33.3.2 Web控件的文档

在本书剩余部分中，你将有机会使用多个ASP.NET Web控件。但你应该花点时间在.NET Framework 4.5 SDK文档中搜索`System.Web.UI.WebControls`命名空间。你可以找到该命名空间下各个成员的说明和代码示例（如图33-7所示）。

System.Web.UI.WebControls... × Manage Content

Class	Description
AccessDataSource	Represents a Microsoft Access database for use with data-bound controls.
AccessDataSourceView	Supports the AccessDataSource control and provides an interface for data-bound controls to perform data retrieval using Structured Query Language (SQL) against a Microsoft Access database.
AdCreatedEventArgs	Provides data for the AdCreated event of the AdRotator control. This class cannot be inherited.
AdRotator	Displays an advertisement banner on a Web page.
AssociatedControlConverter	Provides a type converter that retrieves a list of WebControl controls in the current container.
AuthenticateEventArgs	Provides data for the Authenticate event.
AutoFieldsGenerator	Represents a base class for classes that automatically generate fields for data-bound controls that use ASP.NET Dynamic Data features.
AutoGeneratedField	Represents an automatically generated field in a data-bound control. This class cannot be inherited.
AutoGeneratedFieldProperties	Represents the properties of an AutoGeneratedField object. This class cannot be inherited.

图33-7 .NET Framework 4.5 SDK文档中列出了所有ASP.NET Web控件

33.4 构建ASP.NET汽车网站

由于很多“简单”控件的外观和它们的Windows GUI版本很一致，所以我不会再花时间来枚举基本组件的细节（Button、Label、TextBox等），而是构建一个新网站来演示使用几个更有趣的控件、ASP.NET母版页模型和数据绑定引擎。具体而言，下一个示例会说明如下技术：

- 使用母版页工作；
- 使用站点地图导航；

❑ 使用GridView控件工作；

❑ 使用Wizard控件工作。

首先，创建一个名为AspNetCarsSite的Empty Web Site项目。注意，我们没有创建新的ASP.NET Web Site项目，因为这会添加很多我们还没介绍的启动文件。在本项目中，我们将手动添加所需要的东西。

33.4.1 使用ASP.NET母版页工作

许多网站的多个页面都有一致的外观（一个公共菜单导航系统、通用的标题和页脚内容、公司标志等）。母版页不只是使用*.master文件扩展的ASP.NET页面。母版页对于客户端浏览器不是可见的（事实上，ASP.NET运行时将不服务于这类Web内容）。母版页在站点内定义一个通用的、共享于所有页面的UI布局（或者页面的子集）。

同样，*.master页面会定义各种内容占位符区域来创建一块能让其他*.aspx文件插入的区域。我们会看到，将内容插入到母版页面的*.aspx文件的外观和我们现在研究的*.aspx文件不同。准确地说，这种形式的*.aspx文件叫做内容页面。内容页面是没有定义HTML <form>元素的*.aspx文件（这是母版页面的事情）。

然而，就最终用户而言，请求指向给定的*.aspx文件。在Web服务器端，相关的*.master文件和相关的*.aspx内容页面会合并为一个统一的HTML页面声明。

为了演示母版页面和内容页面的使用，首先通过Website→Add New Item菜单项将一个新的母版页面插入到我们的网站（图33-8显示了结果对话框）。

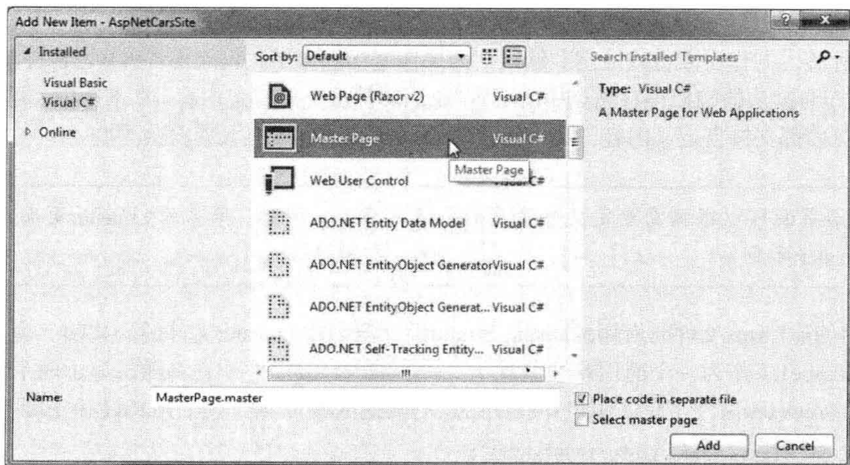


图33-8 插入新的*.master文件

MasterPage.master文件的最初标记差不多如下：

```
<%@ Master Language="C#" AutoEventWireup="true"
    CodeFile="MasterPage.master.cs" Inherits="MasterPage" %>

<!DOCTYPE html>
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title> </title>
  <asp:ContentPlaceHolder id="head" runat="server">
  </asp:ContentPlaceHolder>
</head>
<body>
  <form id="form1" runat="server">
  <div>
    <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
    </asp:ContentPlaceHolder>
  </div>
  </form>
</body>
</html>
```

需要关注的第一点是新的<%@Master%>指令。通常，这个指令和第32章讲述的<%@Page%>支持同样的特性。同Page类型相似，母版页继承自一个特定的基类，在这个例子中是MasterPage。如果要打开相关的代码文件，将会看到下面的类定义：

```
public partial class MasterPage : System.Web.UI.MasterPage
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
}
```

另一个关注点是<asp:ContentPlaceHolder>类型。这个母版页的区域描绘了相关*.aspx文件的UI窗口控件，而不是母版页自己的内容。

如果你确实想要将一个*.aspx文件插入到这个区域中，<asp:ContentPlaceHolder>和</asp:ContentPlaceHolder>标签内的作用域将被清空。然而，如果你这样选择了，就能用各种Web控件填充这个区域，这些Web控件作为默认的UI使得在站点中的给定*.aspx文件不支持特定内容。在本例中，假设每一个在站点中的*.aspx页面都支持自定义内容，那么<asp:ContentPlaceHolder>元素将会为空。

说明　*.master页面可以根据需要定义任意多的内容占位符。同时，单独的*.master页面可以嵌入更多的*.master页面。

你能使用创建*.aspx文件时所用的Visual Studio设计器创建*.master文件的公共UI。为你的站点添加一个描述性的Label（作为一个通用欢迎消息）、一个AdRotator控件（它将随机显示两个图片文件中的一个）和一个TreeView控件（允许用户导航到站点的其他区域）。下面是用IDE设计完母版页后的标记：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title> </title>
    <asp:ContentPlaceHolder id="head" runat="server">
    </asp:ContentPlaceHolder>
</head>
<body>
    <form id="form1" runat="server">
    <div>
      <hr />
```

```
        <asp:Label ID="Label1" runat="server" Font-Size="XX-Large"
          Text="Welcome to the ASP.NET Cars Super Site!"></asp:Label>
        <asp:AdRotator ID="myAdRotator" runat="server"/>
         <br />
        <br />
        <asp:TreeView ID="navigationTree" runat="server">
        </asp:TreeView>
        <hr />
      </div>
      <div>
          <asp:ContentPlaceHolder id="ContentPlaceHolder1" runat="server">
          </asp:ContentPlaceHolder>
      </div>
      </form>
  </body>
  </html>
```

图33-9显示了当前母版页面的设计时视图（注意此时AdRotator的显示区域是空的）。

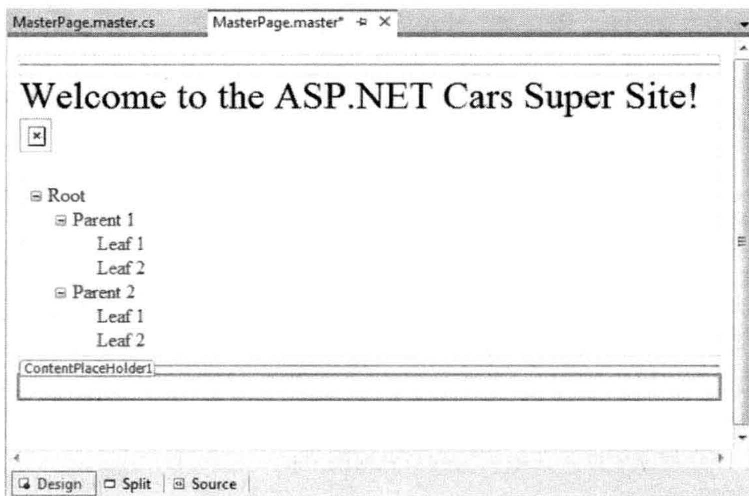


图33-9 *.master文件共享的UI

你可以使用控件内嵌的编辑器并选择Auto Format...链接来强化TreeView控件的外观。同样，也可以使用Properties编辑器美化其他控件。在你满意之后再阅读下面的内容。

1. 使用TreeView控件站点导航逻辑

ASP.NET附带了几个能让我们处理站点导航的Web控件：SiteMapPath、TreeView和Menu。就像我们期望的那样，这些Web组件可以以多种方式进行配置。例如，每一个控件都可以通过外部XML文件（或基于XML的*.sitemap文件）、在代码中以编程方式或通过Visual Studio的设计器标记动态生成其节点。

我们的导航系统可以使用*.sitemap文件动态填充。这个方式的优势是我们可以在外部文件中定义网站的整体结构，并且可以直接把它绑定到TreeView（或Menu）组件。这样，如果网站的导航结构改变了，我们只需要改变*.sitemap文件并且重新加载页面即可。首先，使用Website→Add New Item菜单项打开图33-10所示的对话框，插入一个新的Web.sitemap文件到项目中。

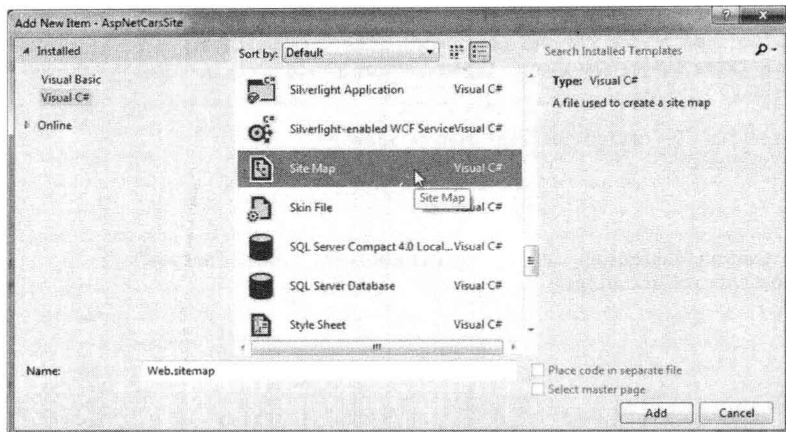


图33-10　插入新的Web.sitemap文件

我们可以看到，最初的Web.sitemap文件定义的最顶层的项具有两个子节点，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="" description="">
    <siteMapNode url="" title="" description="" />
    <siteMapNode url="" title="" description="" />
  </siteMapNode>
</siteMap>
```

如果把这个结构绑定到Menu控件的话，我们就会发现最顶层的项有两个子项。因此，如果我们希望定义子项，就只需要在既有的<siteMapNode>区域中定义新的<siteMapNode>元素。不管怎么样，目标是在Web.sitemap文件中使用各种<siteMapNode>元素定义网站的整体结构。每一个元素都可以定义一个标题和URL特性。URL特性表示用户单击某个菜单项（或TreeView节点）时应该导航到哪个*.aspx文件。我们的网站包含3个站点地图节点（在最顶层的站点地图节点下），设置如下：

- 主页面Default.aspx；
- 构建汽车BuildCar.aspx；
- 查看库存Inventory.aspx。

我们稍后将在项目中添加这三个新的ASP.NET网页。现在，先来简单地配置站点地图文件。

我们的导航有一个最顶层的Welcome项和3个子元素。因此，我们可以按如下所示更新Web.sitemap文件。要知道，每一个url值必须是唯一的（否则，就会收到运行时错误）：

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode url="" title="Welcome!"  description="">
    <siteMapNode url="~/Default.aspx" title="Home"
      description="The Home Page" />
    <siteMapNode url="~/BuildCar.aspx" title="Build a car"
      description="Create your dream car" />
    <siteMapNode url="~/Inventory.aspx" title="View Inventory"
      description="See what is in stock" />
  </siteMapNode>
</siteMap>
```

说明　url特性中每一个页面之前的~/前缀表示网站的根。

现在，尽管我们没有使用给定属性来把Web.sitemap文件直接关联到Menu或TreeView控件上包含要显示Web.sitemap的UI组件的*.master或*.aspx页面必须包含SiteMapDataSource组件。这个组件会在请求页面的时候把Web.sitemap文件自动加载到其对象模型中。然后设置Menu和TreeView类型的DataSourceID属性来指向SiteMapDataSource实例。

如果要新增一个SiteMapDataSource到*.master文件，然后自动设置DataSourceID属性，我们可以使用Visual Studio设计器。激活TreeView控件的内联编辑器（即单击TreeView右上角的小箭头），展开Choose Data Source下拉列表，然后选择New Data Source，如图33-11所示。

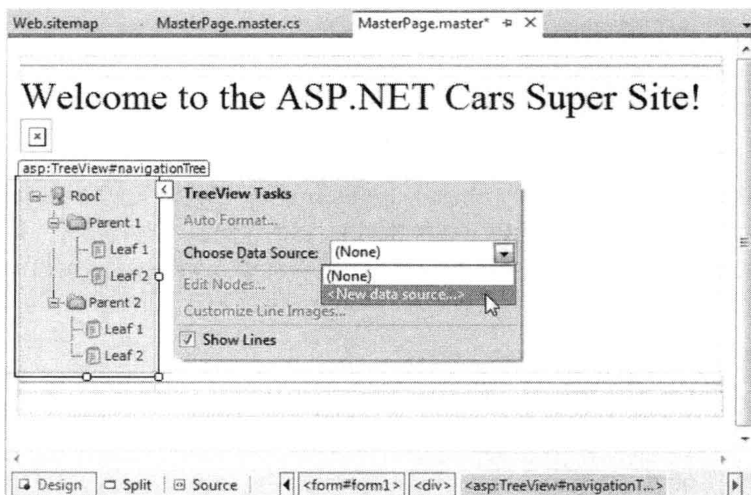


图33-11　新增SiteMapDataSource

从结果对话框中选择SiteMap图标。这会设置Menu或TreeView控件的DataSourceID属性，并且向我们的页面新增一个SiteMapDataSource组件。这就是配置TreeView控件来导航到网站其他页面需要做的所有事情。如果希望在用户选择某个菜单项的时候进行其他处理，可以通过处理TreeView控件的SelectedNodeChanged事件来实现。对于本例，不需要这么做，但是要知道，你可以使用传入的事件参数来检测哪个菜单项被单击了。

2. 使用SiteMapPath类型创建浏览路径

在转移到AdRotator组件之前，向*.master文件的内容占位符元素下增加一个SiteMapPath类型（位于Toolbox中Navigation标签页下）。这个组件会根据菜单系统的当前选择自动调整其内容。你可能知道，这为终端用户提供了有用的可视化提示（正式地说，这个UI技术称为浏览路径）。在完成了这个示例之后，如果选择Welcome→Build a Car菜单项，SiteMapPath组件也就自动进行了相应的更新。

3. 配置AdRotator控件

ASP.NET AdRotator部件的作用是在浏览器的某个位置随机显示一个特定的图片。这时，AdRotator展示为一个空的占位符。从功能上讲，这个控件不能发挥它的作用，除非设置AdvertisementFile属性

指向描述每个图片的源文件。对于这个例子来说，数据源将是一个名为Ads.xml的简单的XML文件。

要在网站中添加了这个XML文件需打开Website→Add New Item菜单项并选择XML文件。将该文件命名为Ads.xml并为每个希望显示的图片指定唯一的<Ad>元素。至少，每个<Ad>元素指定要显示的图片（ImageUrl）、要导航的URL[如果这个图片被选择了（TargetUrl）]、鼠标置于其上时显示的文本内容（AlternateText）、广告出现频率权数（Impressions）：

```
<Advertisements>
  <Ad>
    <ImageUrl>SlugBug.jpg</ImageUrl>
    <TargetUrl>http://www.Cars.com</TargetUrl>
    <AlternateText>Your new Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>car.gif</ImageUrl>
    <TargetUrl>http://www.CarSuperSite.com</TargetUrl>
    <AlternateText>Like this Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
</Advertisements>
```

在这里，我们指定了两个图片文件（slugbug.jpg和car.gif）。因此，我们需要确保这些文件在网站的根目录中（这些文件可以从本书的代码下载中找到）。要把它们加入当前项目，只需要选择WebSite→Add Existing菜单项。此刻，可以通过AdvertisementFile属性（在Properties窗口中）将XML文件与AdRotator控件关联起来，如下所示：

```
<asp:AdRotator ID="myAdRotator" runat="server"
               AdvertisementFile="~/Ads.xml"/>
```

稍后运行这个应用程序并且回传到页面时，将随机展示两个图片文件中的一个。

33.4.2　定义默认的内容页面

现在你已拥有一个创建好了的母版页，可以开始设计单独的*.aspx页面了，这些页面将定义合并在母版页的<asp:ContentPlaceHolder>标签内的UI内容。被合并到母版页的*.aspx文件称为*内容页面*，它与独立的ASP.NET Web页面有一些关键的不同之处。

简言之，*.master文件定义了最后的HTML页面的<form>片段。因此，*.aspx文件内现有的<form>区域将需要由<asp:Content>作用域代替。你可以手动更新初始*.aspx文件的标记，然后将一个新的内容页面自动插入到项目中，不过只需要右击*.master文件设计器界面中的任意地方，然后选择Add Content Page菜单项就可以了（如图33-12所示）。

这就会生成一个具有下面初始标记的新*.aspx文件：

```
<%@ Page Language="C#" MasterPageFile="~/MasterPage.master"
         AutoEventWireup="true" CodeFile="Default.aspx.cs"
         Inherits="_Default" Title=" " %>
<asp:Content ID="Content1"
  ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2"
  ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">
</asp:Content>
```

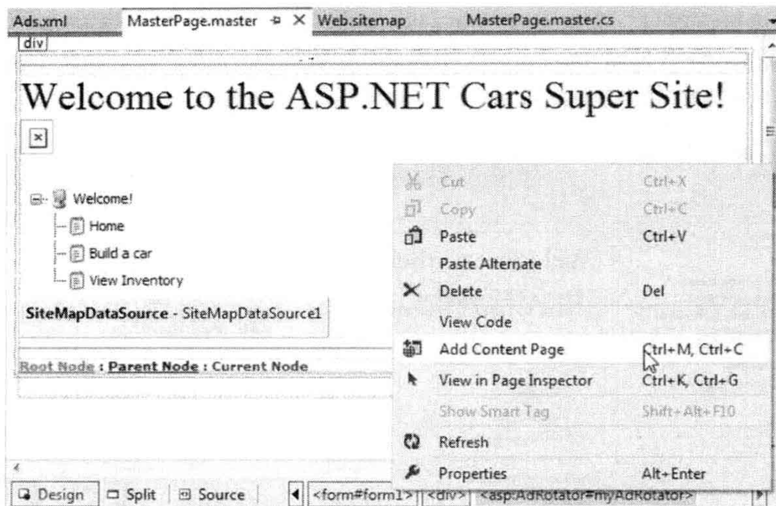


图33-12　在母版页中添加一个新的内容页

首先，注意使用了一个新的被赋值指向*.master文件的MasterPageFile特性，更新了<%@Page%>指令。同时注意有一个<asp:Content>作用域（而非<form>元素），目前为空，它设置的ContentPlaceHolderID值与母版文件内的<asp:ContentPlaceHolder>部件是一样的。

虽然在内容页面上母版的内容以只读形式进行显示，但是有了这样的关联，内容页面就知道要在哪里插入其内容。不需要为Default.aspx内容区域构建复杂的UI，因此对于本例，只需要增加一些文本来提供一些基本的站点指令，如图33-13所示（还要注意设计器中内容页面的右上部分有一个链接可以切换到关联的母版文件）。

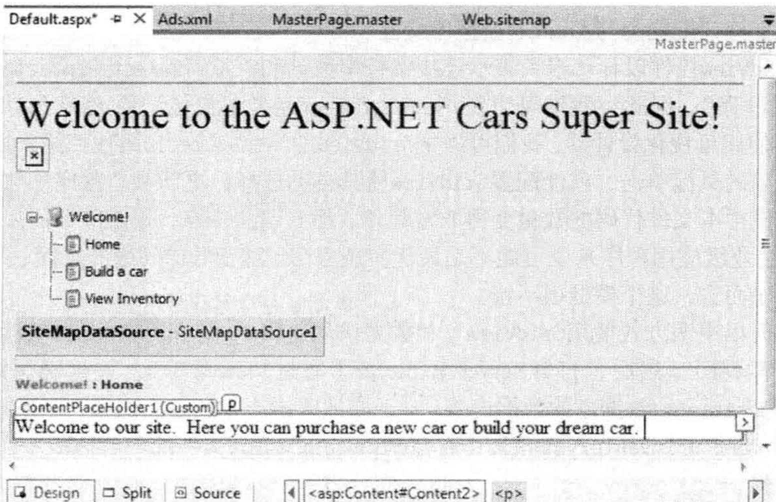


图33-13　创建第一个内容页面

现在，如果运行该项目，你将发现*.master和Default.aspx文件的UI内容已经合并成一个简单的HTML流。如在图33-14所看见的，终端用户无法察觉母版页的存在(浏览器只是简单地展示Default.aspx中的HTML)。按F5键刷新页面，AaRotator将会随机显示两个图片中的一个。

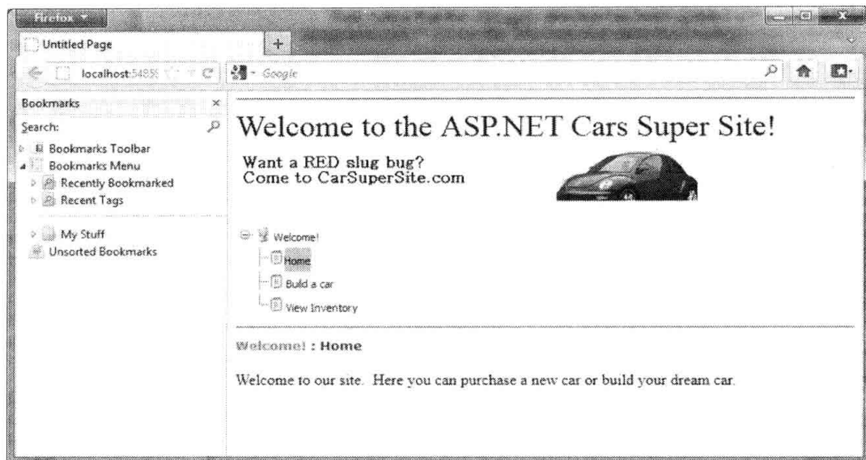


图33-14　在运行时，母版页面和内容页面形成了一个表单

33.4.3　设计Inventory内容页面

为了向当前项目中插入Inventory.aspx内容页面，在IDE中打开*.master页面，然后选择Website→Add Content Page并用Solution Explorer将文件重命名为Inventory.aspx。Inventory内容页面的作用是在一个GridView控件内显示AutoLot数据库的Inventory表的内容。不过与上一章不同的是，GridView将使用内嵌的数据绑定支持，配置为与AutoLot数据库交互。

ASP.NET GridView控件以标记形式表示连接字符串数据和SQL的Select、Insert、Update和Delete语句(或者存储过程)。因此，我们可以使用SqlDataSource类生成标记而无需手动编写所有必要的ADO.NET代码。使用可视化设计器，我们可以把GridView的DataSourceID属性赋给SqlDataSource组件。

使用几个简单的鼠标单击，就能配置GridView使其自动选择、更新和删除底层数据存储的记录。虽然这个零编码使样本文件代码的数量变得非常简单，但要明白一点，伴随简单而来的是控件性能的降低，并且对于企业级应用程序来说可能不是最优实现方法。对于低访问量的页面、网站的原型制作、小的内部应用程序而言，这个模型很不错。

为了演示如何以声明方式使用GridView(和数据访问逻辑)，首先需要使用描述性Label控件更新Inventory.aspx内容页面。然后，打开Server Explorer(通过View菜单)，并且确保添加了我们研究ADO.NET时创建的AutoLot数据库的数据连接(对于增加数据连接的整个过程，可参见第21章)。现在，选择 Server Explorer中的Inventory表并且将它拖到Inventory.aspx文件的内容区域中。完成后，IDE就会进行如下的步骤。

(1) web.config文件已经使用新的<connectionStrings>元素进行更新了。

(2) 一个SqlDataSource组件已经配置了必要的Select、Insert、Update和Delete逻辑。

(3) GridView的DataSourceID属性已经被设置为新的SqlDataSource组件。

说明 作为将表拖曳到Server Explorer的替代方式，我们可以使用内联编辑器（右边）来配置GridView组件。从Choose Data Source下拉框中选择 New Data Source。这会激活一个向导，它通过一系列步骤来把这个组件连接到必要的数据源。

如果研究GridView控件开始的声明，将会发现DataSourceID属性已经被设置为我们刚才定义的SqlDataSource，如下所示：

```
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="False"
     DataKeyNames="CarID" DataSourceID="SqlDataSource1"
     EmptyDataText="There are no data records to display.">
  <Columns>
    <asp:BoundField DataField="CarID" HeaderText="CarID" ReadOnly="True"
         SortExpression="CarID" />
    <asp:BoundField DataField="Make" HeaderText="Make" SortExpression="Make" />
    <asp:BoundField DataField="Color" HeaderText="Color" SortExpression="Color" />
    <asp:BoundField DataField="PetName" HeaderText="PetName"
                    SortExpression="PetName" />
  </Columns>
</asp:GridView>
```

SqlDataSource元素是活动发生的主要地方。注意，在其后的标记中，这个元素记录了必要的SQL语句（不少参数化查询）来和AutoLot数据库的Inventory表进行交互。同样，使用ConnectionString属性的$语法，这个组件可以自动从web.config中读取<connectionStrings>值：

```
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
 ConnectionString="<%$ ConnectionStrings:AutoLotConnectionString1 %>"
  DeleteCommand="DELETE FROM [Inventory] WHERE [CarID] = @CarID"
  InsertCommand="INSERT INTO [Inventory] ([CarID], [Make], [Color], [PetName])
    VALUES (@CarID, @Make, @Color, @PetName)"
  ProviderName="<%$ ConnectionStrings:AutoLotConnectionString1.ProviderName %>"
  SelectCommand="SELECT [CarID], [Make], [Color], [PetName] FROM [Inventory]"
  UpdateCommand="UPDATE [Inventory] SET [Make] = @Make,
    [Color] = @Color, [PetName] = @PetName WHERE [CarID] = @CarID">
  <DeleteParameters>
    <asp:Parameter Name="CarID" Type="Int32" />
  </DeleteParameters>
  <UpdateParameters>
    <asp:Parameter Name="Make" Type="String" />
    <asp:Parameter Name="Color" Type="String" />
    <asp:Parameter Name="PetName" Type="String" />
    <asp:Parameter Name="CarID" Type="Int32" />
  </UpdateParameters>
  <InsertParameters>
    <asp:Parameter Name="CarID" Type="Int32" />
    <asp:Parameter Name="Make" Type="String" />
    <asp:Parameter Name="Color" Type="String" />
    <asp:Parameter Name="PetName" Type="String" />
  </InsertParameters>
</asp:SqlDataSource>
```

此刻，可以运行Web程序了，单击View Inventory菜单项，然后查看数据，如图33-15所示。（还要注意，我用统一的外观通过内联设计器更新GridView网格。）

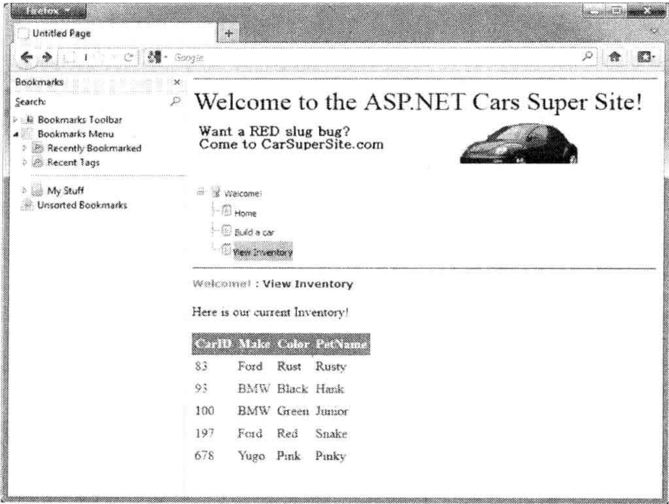


图33-15　无代码的SqlDataSource组件

1. 启用排序和分页

GridView控件能轻松地为排序（通过列名超链接）和分页（通过数字或下一页/上一页超链接）进行配置。如果要这么做，激活内联编辑器，选择适当的选项，如图33-16所示。

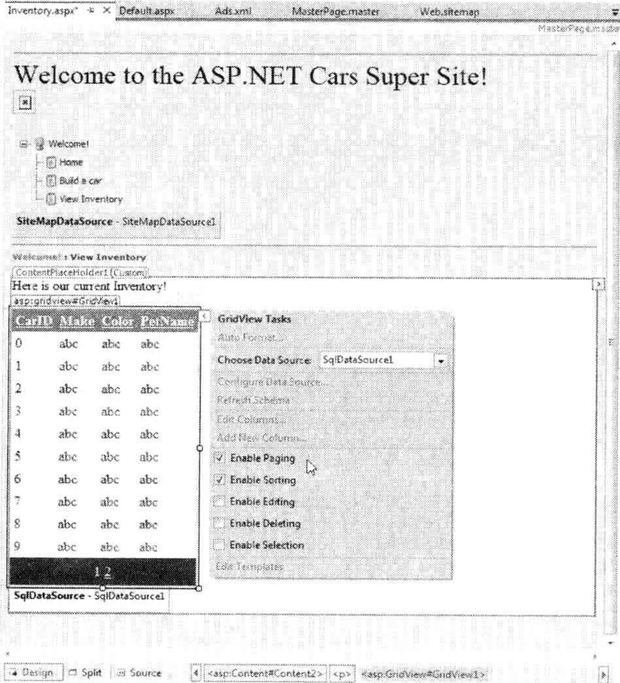


图33-16　启动分页和排序

当再次运行页面时，经由分页链接（假设在Inventory表里有足够的记录）通过单击列名并从头至尾滚动数据，你就能够对数据进行排序了。

2. 启用现场编辑

这个页面的最后一个细节是启用GridView控件的现场编辑功能。由于SqlDataSource已经有了必需的Delete和Update逻辑，所以我们只需要检查GridView的Enable Deleting和Enable Editing复选框（相关的位置请参考图33-16）。可以肯定的是，如果导航回Inventory.aspx页面的话，我们就可以编辑和删除记录(如图33-17所示)，还可以更新AutoLot数据库的底层Inventory表。

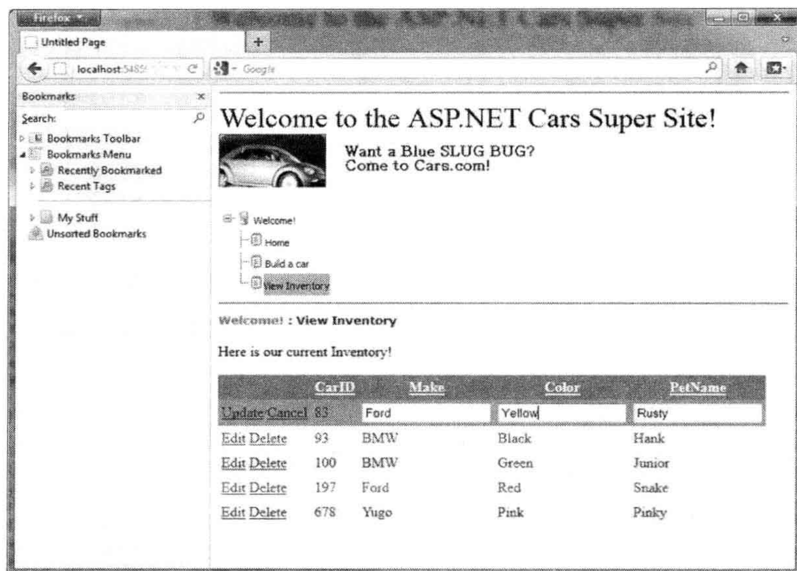


图33-17 编辑和删除功能

说明 要为GridView启用现场编辑，需要为数据库表设置主键。如果看到这些选项没有启用，很可能是因为你忘了将AutoLot数据库Inventory表中的CarID设置为主键。

33.4.4 设计Build-a-Car内容页面

这个例子的最后任务是设计BuildCar.aspx内容页面。打开*.master文件，将文件插入到当前项目（通过Website→Add Content Page菜单项）。还可以右键单击项目的母版页。使用Solution Explorer将新文件改名为BuildCar.aspx。

这个新页面将利用ASP.NET的Wizard Web控件，这个控件提供了一个简单的方法，使得我们通过一系列相关步骤就可以得到终端用户界面。本例中的步骤将模拟创建一个购买汽车的行为。

向内容区域放置一个描述性的Label和Wizard控件。接下来，激活Wizard的内联编辑器并单击Add/Remove WizardSteps链接。一共添加4步，如图33-18所示。

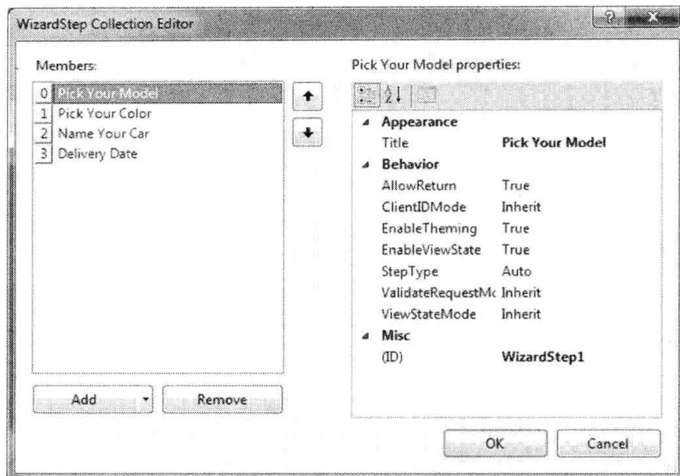


图33-18　配置向导

定义这些步骤之后，你将注意到，Wizard定义了一个空的内容区域。在这个区域里，你现在能够为当前所选择的步骤拖放控件。对于这个例子来说，使用下面的UI元素更新每一步（确定使用Properties窗口为每项提供一个合适的ID值）。

- Pick Your Model：TextBox控件。
- Pick Your Color：ListBox控件。
- Name Your Car：TextBox控件。
- Delivery Date：Calendar控件。

ListBox控件是Wizard的唯一UI元素，Wizard需要另外一些步骤。在设计器上选择此项（确定你首先选择了Pick Your Color链接），然后通过Properties窗口的Items属性使用一套颜色填充这个部件。这时你将发现，标记与下面在Wizard定义作用域内的内容非常相似：

```
<asp:ListBox ID="ListBoxColors" runat="server" Width="237px">
  <asp:ListItem>Purple</asp:ListItem>
  <asp:ListItem>Green</asp:ListItem>
  <asp:ListItem>Red</asp:ListItem>
  <asp:ListItem>Yellow</asp:ListItem>
  <asp:ListItem>Pea Soup Green</asp:ListItem>
  <asp:ListItem>Black</asp:ListItem>
  <asp:ListItem>Lime Green</asp:ListItem>
</asp:ListBox>
```

既然已经定义了每一步，你就能够为自动生成的Finish按钮处理FinishButtonClick事件了。但要注意的是，直到在设计器中选择向导的最后一步之后，才会看到这个Finish按钮。一旦选择了最后一步，可以双击Finish按钮来生成事件处理程序。在服务器端的事件处理程序内，从每个UI元素中获得选择项并创建一个描述字符串，这个字符串被分配给了另外一个名为lblOrder的Label类型的Text属性，如下所示：

```
public partial class BuildCarPage : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
```

```
    {
    }

    protected void carWizard_FinishButtonClick(object sender,
      WizardNavigationEventArgs e)
    {
      // 得到每一个值
      string order = string.Format("{0}, your {1} {2} will arrive on {3}.",
      txtCarPetName.Text, ListBoxColors.SelectedValue,
      txtCarModel.Text,
      carCalendar.SelectedDate.ToShortDateString());

      // 赋值给标签
      lblOrder.Text = order;
    }
  }
```

现在AspNetCarsSite就完成了。图33-19显示了运行中的Wizard。

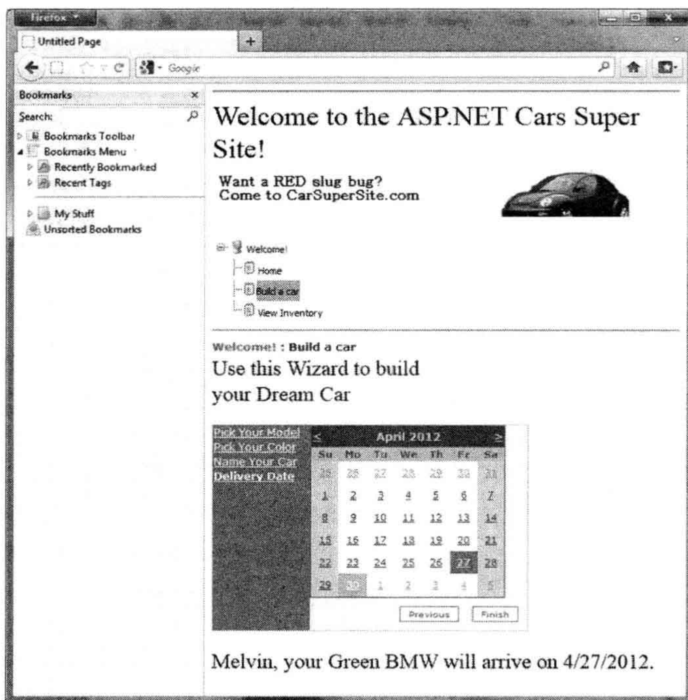


图33-19 运行中的Wizard

至此，对各种ASP.NET Web控件、母版页、内容页和站点地图导航的研究就结束了。接下来，我们看看ASP.NET验证控件的功能。为了使本章话题保持独立，我们将构建一个新的网站来演示验证技术。不过，你很可能已经在当前项目中添加了验证控件。

源代码 AspNetCarsSite网站的源代码位于Chapter 33子目录下。

33.5　验证控件的作用

我们将研究的下一组Web窗体控件统称为验证控件（validation control）。不同于已经研究过的其他Web窗体控件，验证控件并不是用来生成HTML的，而是通过客户端JavaScript实现窗体验证的目的。本章开始已经提到了，客户端窗体验证是非常有用的，因为你能确保在回传到Web服务器前，将各种各样的约束放置在适当的位置上，从而避免代价高昂的往返行程。表33-3简要列出了一些ASP.NET验证控件。

表33-3　ASP.NET验证控件

控　　件	作　　用
CompareValidator	验证一个输入控件的值是否等于另一个输入控件或固定常量的给定值
CustomValidator	允许创建一个自定义的验证功能，以验证给定的控件
RangeValidator	在一个预先确定的范围内确定一个给定值
RegularExpressionValidator	检查相关联的输入控件的值是否匹配一个正则表达式的模式
RequiredFieldValidator	确保一个给定输入控件包含有一个值（即非空）
ValidationSummary	使用一个列表、一个项目列表或一个段落显示网页上所有验证错误的概况。这些错误可以在一行内显示或在弹出的信息框里显示

所有的验证控件都派生于名为System.Web.UI.WebControls.BaseValidator的公共基类（ValidationSummary除外）。因此它们拥有一套公共的特征。表33-4记载了其中关键的成员。

表33-4　ASP.NET验证程序的公共属性

成　　员	作　　用
ControlToValidate	获取或设置输入控件以进行验证
Display	获取或设置验证控件里的出错信息的显示方式
EnableClientScript	获取或设置指明客户端验证是否启动的值
ErrorMessage	获取或设置出错信息的文本
ForeColor	获取或设置当验证失败时信息的颜色

为了演示如何使用验证控件，我们创建一个名为ValidatorCtrls的空网站项目并向其中插入一个名为Default.aspx的新Web表单。首先，在页面上放置4个TextBox控制（带有4个对应的、描述性的Label）。接下来，在邻近每个输入字段处放置RequiredFieldValidator、RangeValidator、RegularExpressionValidator和CompareValidator控件。最后，添加一个Button和Label。图33-20展示了一种可能的布局。

现在已经有一个初始UI，让我们完成配置所有验证器控件的全过程并查看最终的结果。不过在此之前，我们需要修改当前的web.config文件，来允许客户端处理验证控件。

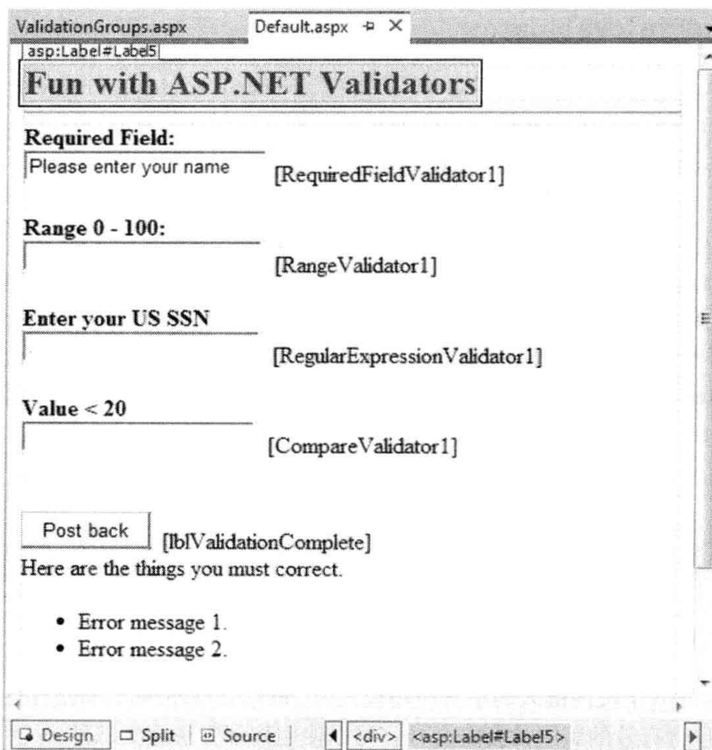


图33-20 ASP.NET验证控件保证在回传前表单数据是正确的

33.5.1 开启客户端JavaScript验证支持

从ASP.NET 4.5开始，微软引入了新的设置，可以在运行时控制验证控件的响应。打开web.config文件，可以找到下面的设置：

```
<appSettings>
...
  <add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
</appSettings>
```

如果配置文件中包含这段配置，那么网站将使用不同的HTML 5数据特性来处理验证，而不是发回客户端JavaScript代码再由浏览器进行处理。由于本书不会深入HTML 5的细节，我们需要将这一行注释掉（或移除），以使当前的验证示例正确工作。因此，简单地注释掉<appSettings>节中的该节点：

```
<appSettings>
...
  <!--
  <add key="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
  -->
</appSettings>
```

33.5.2　RequiredFieldValidator

配置RequiredFieldValidator很简单。只需要使用Visual Studio的Properties窗口相应地设置ErrorMessage和ControlToValidate属性。产生的标记能确保txtRequiredField文本框不为空：

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
  runat="server" ControlToValidate="txtRequiredField"
  ErrorMessage="Oops!  Need to enter data.">
</asp:RequiredFieldValidator>
```

RequiredFieldValidator支持InitialValue属性。可以使用这个属性确保用户在相关的TextBox中输入任何不同于初始值的值。例如，当用户首次登录到一个页面时，你可能希望配置一个包含"Please enter your name"值的文本框。现在，如果不设置RequiredFieldValidator的InitialValue属性，运行时将认为字符串"Please enter your name"是有效的。这样，为了确保只有当用户输入任何非"Please enter your name"的内容时TextBox才有效，按如下代码所示配置部件：

```
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
  runat="server" ControlToValidate="txtRequiredField"
  ErrorMessage="Oops!  Need to enter data."
  InitialValue="Please enter your name">
</asp:RequiredFieldValidator>
```

33.5.3　RegularExpressionValidator

当希望对在给定输入字段内键入的字符应用模式时，RegularExpressionValidator很有用。为了确保一个给定Textbox包含有效的美国社会安全号，可以使用如下方法定义部件：

```
<asp:RegularExpressionValidator ID="RegularExpressionValidator1"
  runat="server" ControlToValidate="txtRegExp"
  ErrorMessage="Please enter a valid US SSN."
  ValidationExpression="\d{3}-\d{2}-\d{4}">
</asp:RegularExpressionValidator>
```

注意，RegularExpressionValidator是如何定义ValidationExpression属性的。如果你以前从未使用过正则表达式，对这个例子需要关心的就是正则表达式用来匹配一个给定的字符串模式。这里，表达式"\d{3}-\d{2}-\d{4}"表示获取一个结构为xxx-xx-xxxx（这里x表示任何阿拉伯数字）的标准美国社会安全号码。

这个特殊的正则表达式比较容易理解，然而，如果你希望测试一个有效的日本电话号码，正确的表达式就变得复杂得多了："(0\d{1,4}-|\(0\d{1,4}\) ?)?\d{1,4}-\d{4}"。好消息是当使用Properties窗口选择ValidationExpression属性时，你能从一套预定义的公共正则表达式中选择。

说明　如果对正则表达式感兴趣，你将高兴地了解到.NET平台提供两个命名空间（System.Text.RegularExpressions和System.Web.RegularExpressions），专用于这类模式的程序处理。

33.5.4　RangeValidator

除了MinimumValue和MaximumValue属性外，RangeValidator还有一个名为Type的属性。如果你想测

试用户输入的值是否在某两个整数之间，需要指定Integer（该项非默认），如下所示：

```
<asp:RangeValidator ID="RangeValidator1"
  runat="server" ControlToValidate="txtRange"
  ErrorMessage="Please enter value between 0 and 100."

  MaximumValue="100" MinimumValue="0" Type="Integer">
</asp:RangeValidator>
```

RangeValidator也能够用于测试给定值是否在一个货币值、日期、浮点数或者字符串数据之间（默认的设置）。

33.5.5 CompareValidator

最后，注意CompareValidator支持下面的Operator属性：

```
<asp:CompareValidator ID="CompareValidator1" runat="server"
  ControlToValidate="txtComparison"
  ErrorMessage="Enter a value less than 20." Operator="LessThan"
  ValueToCompare="20" Type="Integer">
</asp:CompareValidator>
```

上面这个验证程序的作用是，使用二元操作符将文本框内的值和另一个值进行比较，Operator属性设置为像LessThan、GreaterThan、Equal和NotEqual那样的值就不会让人感到惊讶了。同时注意，ValueToCompare用于建立一个用来比较的值。还要注意我们将Type特性设置成了Integer。默认情况下，CompareValidator验证的是字符串值。

说明 CompareValidator也能使用ControlToCompare属性配置，用以比较另一个Web窗体控件（而非一个硬编码值）内的值。

最后完成这个页面的代码，处理Button控件的Click事件，并通知用户他已经成功地进行了验证逻辑，如下所示：

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {

  }
  protected void btnPostback_Click(object sender, EventArgs e)
  {
    lblValidationComplete.Text = "You passed validation!";
  }
}
```

现在，使用你的浏览器导航到该页面。此刻，你不应该看到任何显著的变化。然而，当试图在输入错误数据后单击Submit按钮时，错误信息立刻就显现出来了。如果查看浏览器里显示的HTML源代码，将看见验证控件生成了一个客户端JavaScript函数，这个函数使用自动下载到用户计算机的JavaScript函数的指定库。一旦客户端验证通过，表单数据就回传到服务器，其中ASP.NET运行时将执行在Web服务器上的相同的验证测试（以确保表单数据在传送到服务器的过程中没有人篡改）。

相关提示：如果HTTP请求被一个不支持客户端JavaScript的浏览器发送，所有的验证将会发生在服务器上。使用这种方式就能针对验证控件编程而不必考虑目标浏览器，返回的HTML页面将错误处理重定向到Web服务器。

33.5.6　创建ValidationSummary

我们要研究的下一个与验证相关的话题是ValidationSummary部件的使用。目前，每个验证控件都在准确的位置显示了它的错误信息，这个位置在设计时定位。在许多情况下，这也许正是你正在寻找的东西。然而，在一个有着大量输入部件的复杂窗体上，你可能不希望有随机的红色文本斑点时不时地蹦出来。使用ValidationSummary类型，能够指示所有的验证类型在页面的特定位置处显示它们的错误信息。

第一步是在*.aspx文件上放置一个ValidationSummary。你可以选择设置这个类型的HeaderText属性和DisplayMode属性，默认情况下这个控件使用项目列表列出所有错误信息：

```
<asp:ValidationSummary id="ValidationSummary1"
  runat="server" Width="353px"
  HeaderText="Here are the things you must correct.">
</asp:ValidationSummary>
```

接下来，把页面上每个单独的验证程序（例如，RequiredFieldValidator、RangeValidator等）的Display属性设置为None。这将确保不会出现验证失败的重复错误信息（一个在摘要窗格里，另一个在验证控件的位置）。图33-21显示了运行中的摘要窗格。

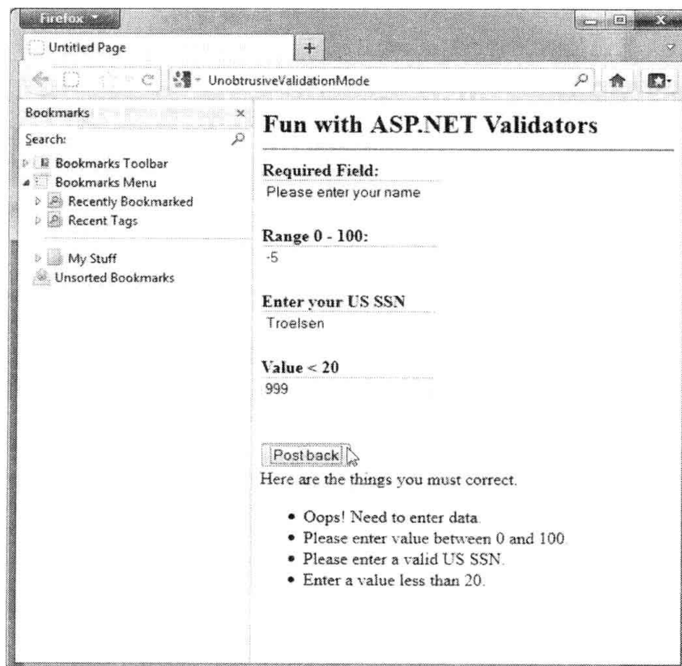


图33-21　使用验证摘要

最后，如果你更愿意使用客户端提示框显示错误信息，可以将ValidationSummary控件的ShowMessageBox属性设置为true，将ShowSummary属性设置为false。

33.5.7 定义验证分组

我们还可以定义验证器程序所属的分组。如果页面区域作为一个整体一起工作的话，这就很有用。例如，在一个Panel对象中，可以有一组控件允许用户输入他或她的邮件地址，另外一个Panel包含了UI元素，用来收集信用卡信息。使用分组，我们可以配置每一个控件组来独立验证。

将一个新的Validation.aspx页面插入到当前项目中，它定义了两个Panel。第一个Panel对象需要有一个TextBox来包含一些用户的输入（通过RequiredFieldValidator），而第二个Panel需要一个美国社会安全号（通过RegularExpressionValidator）。图33-22显示了每一个可能的UI。

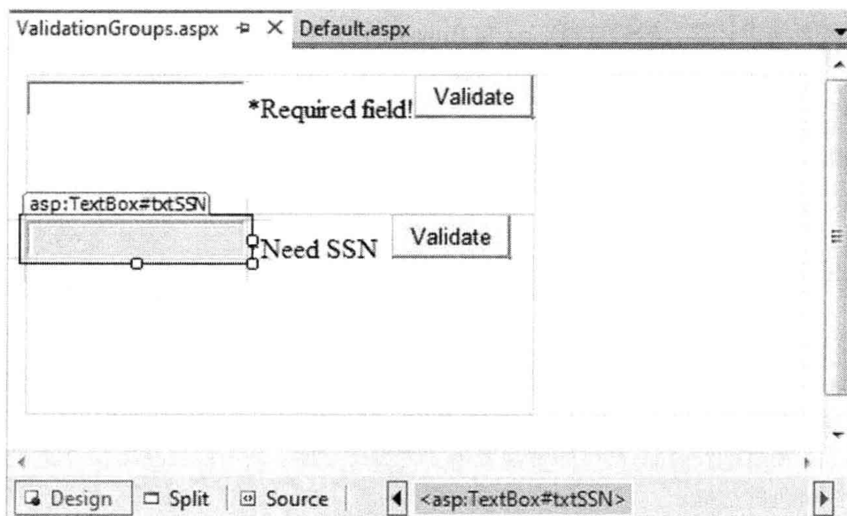


图33-22 这些Panel对象会独立配置它们的输入区域

为了确保这些验证程序独立工作，只需要使用ValidationGroup属性为每一个验证程序和要验证的控件设置唯一的分组名。下面是可能的标记（注意，这里用的Click事件在代码文件中是空的，它们只是用来允许在Web服务器上产生回发）：

```
<form id="form1" runat="server">

  <asp:Panel ID="Panel1" runat="server" Height="83px" Width="296px">
    <asp:TextBox ID="txtRequiredData" runat="server"
            ValidationGroup="FirstGroup">
    </asp:TextBox>
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
         ErrorMessage="*Required field!" ControlToValidate="txtRequiredData"
         ValidationGroup="FirstGroup">
    </asp:RequiredFieldValidator>
    <asp:Button ID="bntValidateRequired" runat="server"
         OnClick="bntValidateRequired_Click"
         Text="Validate" ValidationGroup="FirstGroup" />
```

```
  </asp:Panel>

  <asp:Panel ID="Panel2" runat="server" Height="119px" Width="295px">
    <asp:TextBox ID="txtSSN" runat="server"
        ValidationGroup="SecondGroup">
    </asp:TextBox>
    <asp:RegularExpressionValidator ID="RegularExpressionValidator1"
        runat="server" ControlToValidate="txtSSN"
        ErrorMessage="*Need SSN" ValidationExpression="\d{3}-\d{2}-\d{4}"
        ValidationGroup="SecondGroup">
    </asp:RegularExpressionValidator> 
    <asp:Button ID="btnValidateSSN" runat="server"
        OnClick="btnValidateSSN_Click" Text="Validate"
      ValidationGroup="SecondGroup" />
  </asp:Panel>

</form>
```

现在，右击页面设计器，然后选择View In Browser菜单项，验证每一个面板组件是否以独立的方式进行工作。

源代码　ValidatorCtrls网站的源代码位于Chapter 33子目录下。

33.6　使用主题

至此，我们已经使用了许多ASP.NET Web控件。你已经看到了，每一个控件都公开了一组属性（很多都是从System.Web.UI.WebControls.WebControl继承而来），用来让我们创建某个UI外观（背景色、文字大小、边框样式等）。当然，在多页面的网站中，网站整体为各种类型的控件定义公共的外观是很正常的。例如，所有的TextBox都可能被配置为支持某个字体，所有Button都有自定义的图像，所有Calendar都有浅蓝色的边框。

很明显，为网站每一个页面的每一个部件创建相同的属性的劳动强度很大，而且可能出错。即使手动为每个页面上的每一个UI组件更新属性，想象一下为每一个TextBox改变背景颜色是多么痛苦啊。显然，肯定有更好的方式来应用网站级别的UI设置。

一种可以用来简化应用公共UI外观的方式就是定义*样式表*。如果你有Web开发背景，就知道样式表定义的一组UI相关的设置是应用在浏览器上的。就像你期望的那样，可以通过设置CssStyle属性来为ASP.NET Web控件设置样式。

然而，ASP.NET还提供了另外一种叫做*主题*的技术来定义公共的UI。和样式表不同，主题应用在Web服务器上（而不是浏览器），并且可以以编程或声明方式实现。由于主题应用在Web服务器上，所以它可以访问网站上所有服务器端的资源。此外，通过编写在任何*.aspx文件中能找到的相同标记来定义主题（你可能也会同意，样式表的语法更简洁）。

回忆一下第32章，ASP.NET Web应用程序可以定义许多"特殊的"子目录，其中一个就是App_Theme。这个子目录可以再分成其他子目录，每一个都表示网站上可能的主题。例如，如图33-23所示，它显示了一个包含3个子目录的App_Theme文件夹，每一个都是构成主题的一组文件。

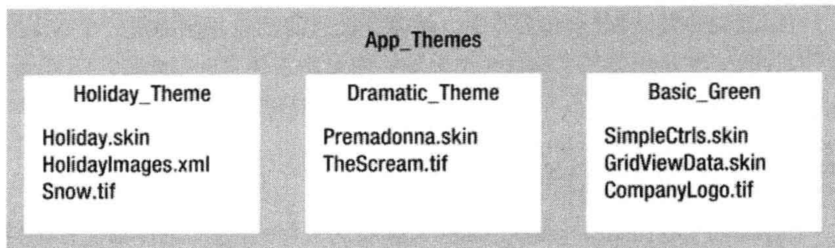


图33-23 一个App_Theme文件夹可能定义许多主题

33.6.1 *.skin文件

每一个主题子目录都必有的文件就是*.skin文件。这些文件定义了各种Web控件的外观。为了举例说明，新建一个Empty Web Site，命名为FunWithThemes，然后插入一个名为Default.aspx的Web Form。在这个页面中，添加`Calendar`、`TextBox`和`Button`控件。对于这些控件，我们不需要特别的配置，而且它们的名称也与当前示例无关。这些控件将作为我们自定义皮肤的对象。

然后，插入一个新的*.skin文件BasicGreen.skin（使用WebSite→Add New Item菜单项），如图33-24所示。

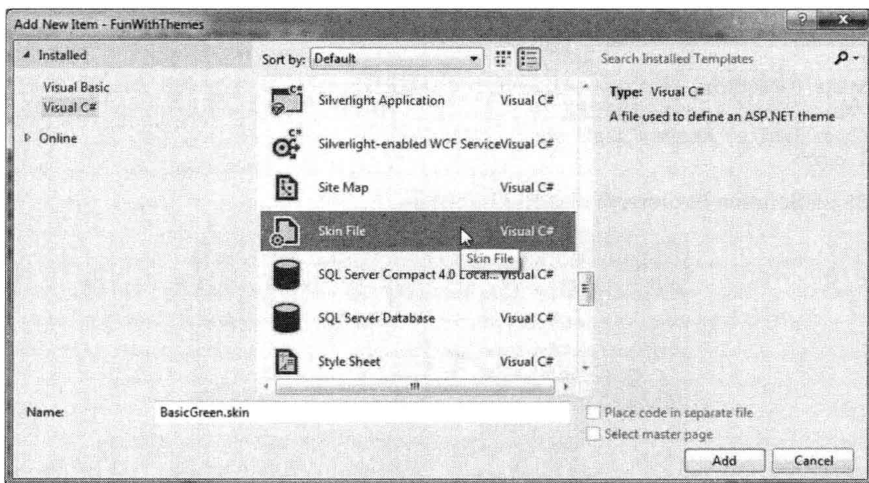


图33-24 插入*.skin文件

Visual Studio会提示你确认这个文件是否可以加到App_Theme文件夹（其实我们就希望这么做）。如果我们现在查看Solution Explorer，确实会发现App_Theme文件夹中的BasicGreen子文件夹包含了新的BasicGreen.skin文件。

回忆一下，*.skin文件是我们使用ASP.NET控件声明语法定义各种组件外观的地方。可惜的是，IDE没有为*.skin文件提供设计器支持。一个减少输入时间的方法就是将一个临时的*.aspx文件插入到程序中（如temp.aspx），它可以用来通过Visual Studio页面设计器构建组件的UI控件。

最后的标记可以复制并粘贴到*.skin文件中。然而，此后我们必须删除每一个Web控件的ID特性。这是有意义的，因为我们不是想为某个Button（例如）而是想为所有Button定义UI外观。

下面是BasicGreen.skin的标记，它为Button、TextBox和Calendar类型定义了默认的外观：

```
<asp:Button runat="server" BackColor="#80FF80"/>
<asp:TextBox runat="server" BackColor="#80FF80"/>
<asp:Calendar runat="server" BackColor="#80FF80"/>
```

注意，每一个控件仍然有runat="server"属性（强制的），并且任何组件都没有ID特性。

现在，让我们定义第二个主题CrazyOrange。使用解决方案资源管理器，右击App_Themes文件夹，并且新增一个CrazyOrange主题。这会在网站的App_Theme文件夹中新增一个子目录。

然后，在Solution Explorer中右击新的CrazyOrange文件夹并且选择Add New项。从结果对话框中新增*.skin文件。更新CrazyOrange.skin文件来为相同的Web控件定义独特的UI外观。例如：

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:TextBox runat="server" BackColor="#FF8000"/>
<asp:Calendar BackColor="White" BorderColor="Black"
  BorderStyle="Solid" CellSpacing="1"
  Font-Names="Verdana" Font-Size="9pt" ForeColor="Black" Height="250px"
  NextPrevFormat="ShortMonth" Width="330px" runat="server">
  <SelectedDayStyle BackColor="#333399" ForeColor="White" />
  <OtherMonthDayStyle ForeColor="#999999" />
  <TodayDayStyle BackColor="#999999" ForeColor="White" />
  <DayStyle BackColor="#CCCCCC" />
  <NextPrevStyle Font-Bold="True" Font-Size="8pt" ForeColor="White" />
  <DayHeaderStyle Font-Bold="True" Font-Size="8pt"
    ForeColor="#333333" Height="8pt" />
  <TitleStyle BackColor="#333399" BorderStyle="Solid"
    Font-Bold="True" Font-Size="12pt"
    ForeColor="White" Height="12pt" />
</asp:Calendar>
```

至此，我们的Solution Explorer窗口如图33-25所示。

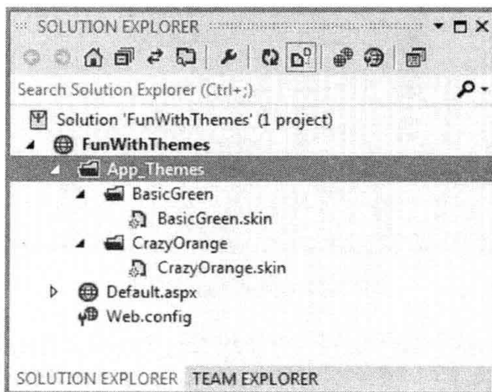


图33-25　一个具有多个主题的网站

既然我们的网站定义了一些主题，下一步就是如何把它们应用到页面上。你可能会想到，这有很多方式。

说明 可以肯定的是，这些示例主题很一般（为了减小篇幅），尽管按照你喜欢的去调整吧。

33.6.2 应用网站级别的主题

如果你希望确保网站的所有页面都遵循相同的主题，最简单的方式就是更新web.config文件。打开当前的web.config文件并且找到<system.web>根元素中的<pages>元素。如果你为<pages>元素增加了主题特性，它就会确保网站每一个页面都分配了所选的主题（当然，就是App_Theme中一个子目录的名字）。下面是关键的更新：

```
<configuration>
  <system.web>
      ...
<pages controlRenderingCompatibilityVersion="4.5"
       theme="BasicGreen">
    </pages>
  </system.web>
</configuration>
```

如果运行该页面，将会发现每个组件都有BasicGreen的UI了。如果将主题特性更新为CrazyOrange并且再次运行这个页面，就会发现使用的是由这个主题定义的UI。

33.6.3 在页面级别应用主题

还可以在页面级别主题，这在很多情况下都很有用。例如，可能web.config文件定义了网站级别的主题（之前描述的），但我们希望为某个页面分配不同的主题。要这么做，我们只需要更新<%@Page%>指令。如果使用Visual Studio的话，就会发现智能感知会显示App_Theme文件夹中定义的每一个主题：

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" Theme ="CrazyOrange" %>
```

由于为这个页面分配了CrazyOrange主题，而web.config文件指定了BasicGreen主题，除了这个页面之外的所有页面都会使用BasicGreen进行呈现。

33.6.4 SkinID属性

有时你会希望为一个组件定义一组可能的UI外观。例如，为CrazyOrange主题中的Button类型定义两个可能的UI。这时你可以使用*.skin文件中的SkinID属性分别设置每一个的外观，如下所示：

```
<asp:Button runat="server" BackColor="#FF8000"/>
<asp:Button runat="server" SkinID = "BigFontButton"
            Font-Size="30pt" BackColor="#FF8000"/>
```

现在，如果有一个页面使用CrazyOrange主题，每一个Button都会被默认赋值为未命名的Button皮肤。如果你希望*.aspx文件中各种按钮都使用BigFontButton皮肤的话，只需要在标记中指定SkinID属性：

```
<asp:Button ID="Button2" runat="server"
            SkinID="BigFontButton" Text="Button" /><br />
```

33

33.6.5　以编程方式分配主题

最后，还可以在代码中分配主题。当希望提供一种方式让最终用户为当前会话选择主题时，这很有用。当然，我们还没有研究如何构建有状态的Web应用程序，因此当前主题的选择会在回发后丢失。在产品级别的网站中，你可能希望把用户当前的主题设置保存在会话变量中或持久化在数据库中。

为了说明如何以编程方式分配主题，我们使用3个新的Button控件更新Default.aspx文件的UI，如图33-26所示。完成后，为每一个Button处理Click事件。

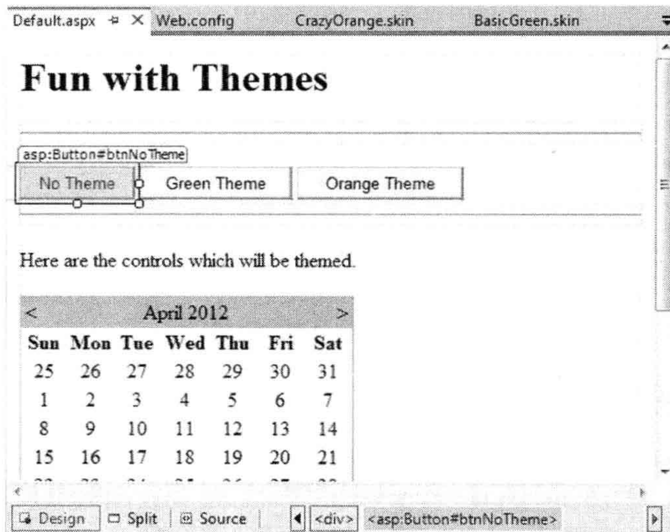


图33-26　主题示例更新后的UI

现在你只可以在页面生命周期的某个阶段以编程方式分配主题。这通常在Page_PreInit事件中完成，按如下所示更新我们的代码文件：

```
partial class _Default : System.Web.UI.Page
{
  protected void btnNoTheme_Click(object sender, System.EventArgs e)
  {
    // 空的字符串导致不应用主题
    Session["UserTheme"] = "";

    // 再一次触发PreInit事件
    Server.Transfer(Request.FilePath);
  }

  protected void btnGreenTheme_Click(object sender, System.EventArgs e)
  {
    Session["UserTheme"] = "BasicGreen";

    // 再一次触发PreInit事件
    Server.Transfer(Request.FilePath);
  }
```

```
    protected void btnOrangeTheme_Click(object sender, System.EventArgs e)
    {
      Session["UserTheme"] = "CrazyOrange";

      // 再一次触发PreInit事件
      Server.Transfer(Request.FilePath);
    }

    protected void Page_PreInit(object sender, System.EventArgs e)
    {
      try
      {
        Theme = Session["UserTheme"].ToString();
      }
      catch
      {
        Theme = "";
      }
    }
  }
```

注意，我们把所选的主题保存在了叫UserTheme的会话变量中（参见第34章），它在Page_PreInit()事件处理程序中被正式赋值。还要注意，当用户单击某个按钮时，我们以编程方式通过调用Server.Transfer()再次请求当前页面来强制触发PreInit事件。如果你运行这个页面，就会发现我们可以通过各种Button单击来创建主题。

源代码 FunWithThemes网站的源代码位于Chapter 33子目录下。

33.7 小结

本章研究了如何使用各种ASP.NET Web控件。我们首先研究了Control和WebControl基类的作用，然后研究了如何与面板的内部控件集合动态交互。随后我们又研究了新的网站导航模型（*.sitemap文件和SiteMapDataSource组件）、新的数据绑定引擎（通过SqlDataSource组件和GridView控件）和各种验证控件。

本章的后半部分研究了母版页和主题的作用。回忆一下，母版页可以为网站的一组页面定义公共的框架。*.master文件定义了许多内容占位符来让内容页插入它们的自定义UI内容。最后，我们了解到，通过ASP.NET主题引擎，可以以声明方式或编程方式在Web服务器端为组件应用公共的UI外观。

第34章

ASP.NET状态管理技术

前两章主要讨论ASP.NET页面的构成和行为，以及它们所包含的控件。本章在此基础上学习Global.asax文件和HttpApplication类型的作用。HttpApplication支持处理大量的事件，而事件则将Web应用程序连接成一个内聚单元，使其不再是一堆彼此独立由母版页驱动的*.aspx文件。

除了探讨HttpApplication类型之外，本章也将讨论非常重要的状态管理话题，如视图状态、会话级变量和应用程序级变量（包括应用程序缓存）、cookie数据和ASP.NET用户配置API的作用。

34.1 状态问题

第32章的开始指出了HTTP是一个*无状态的联网协议*。这个事实使Web开发与构建一个可执行程序集的过程截然不同。例如，在创建Windows桌面UI应用程序时，你能确定任何定义在Form派生类中的成员变量通常将存在于内存中，直到用户显式地关闭可执行程序：

```
public partial class MainWindow : Window
{
  // 状态数据
  private string userFavoriteCar = "Yugo";
}
```

然而，在万维网领域中，不允许提供同样奢侈的假设。为了证明这一点，创建一个新的空网站项目（命名为SimpleStateExample）并且插入一个新的Web表单。在*.aspx文件的代码隐藏文件中，定义一个名为userFavoriteCar的页面级字符串变量，如下所示：

```
public partial class _Default : System.Web.UI.Page
{
  // 状态数据
  private string userFavoriteCar = "Yugo";

  protected void Page_Load(object sender, EventArgs e)
  {
  }
}
```

接下来，构造如图34-1显示的简单Web UI。

服务器端用于Set按钮（名为btnSetCar）的Click事件处理程序将允许用户使用TextBox内的值（名为txtFavCar）分配字符串变量：

```
protected void btnGetCar_Click(object sender, EventArgs e)
{
```

```
    // 在成员变量中存储用户最喜欢的车
    userFavoriteCar = txtFavCar.Text;
}
```

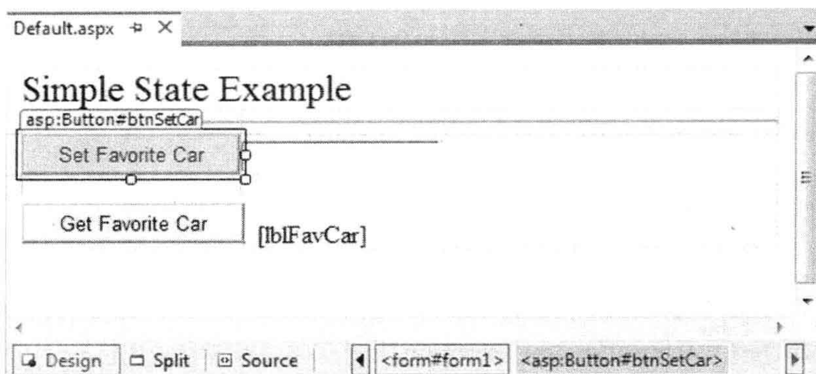


图34-1 简单状态页的UI

而用于Get按钮（btnGetCar）的Click事件处理程序将展示页面Label部件（lblFavCar）内的成员变量的当前值，如下所示：

```
protected void btnGetCar_Click(object sender, EventArgs e)
{
  // 显示成员变量的值
  lblFavCar.Text = userFavoriteCar;
}
```

现在，如果你正在构建一个Windows GUI应用程序，可以假设一旦用户设置了初始值，它将在桌面应用程序的生命期中被保存。遗憾的是，当运行这个Web应用程序时，你将发现，每次回传到Web服务器（通过单击任何按钮），userFavoriteCar字符串变量的值都被设置回初始值“Yugo”，因此，Label文本总是固定值。

由于发送HTTP响应时，HTTP无法自动保存数据，这就是Page对象被立即销毁的原因。因此，当客户端回传到*.aspx文件时，一个新的Page对象即被建立，它将重置任何页面级的成员变量。这无疑是一个进退两难的问题。想象一下，如果在线购物时每次都要回传到Web服务器端，并且先前输入的（例如你想要购买的项目）任何一个或者所有的信息都丢失的话，将是多么痛苦。当希望保存登录到站点的用户信息时，你需要利用各种状态管理技术。

说明 不只是ASP.NET要面对这个问题，JavaWeb应用程序、CGI应用、传统ASP和PHP应用程序都必须面对状态管理这个令人痛苦的问题。

为了在两次回传之间保存userFavoriteCar字符串类型的值，需要在会话变量内存储这个字符串类型的值。在后面我们会去了解会话状态的确切细节。为了完成这个过程，下面提供了当前页必需的更新代码（注意，这里不再使用私有字符串成员变量，因此可以随意注释掉或删除前面定义的字符串变量）：

```
public partial class _Default : System.Web.UI.Page
{
  // 状态数据
  // 私有字符串userFavoriteCar值为“Yugo”

  protected void Page_Load(object sender, EventArgs e)
  {
  }

  protected void btnSetCar_Click(object sender, EventArgs e)
  {
    // 保存将要存储在会话变量中的值
    Session["UserFavCar"] = txtFavCar.Text;
  }

  protected void btnGetCar_Click(object sender, EventArgs e)
  {
    // 获取会话变量值
    lblFavCar.Text = (string)Session["UserFavCar"];
  }
}
```

如果现在运行这个应用程序，你最喜欢的汽车的值将在各次回传中被保存起来，这要归功于HttpSessionState对象，它巧妙利用了继承的Session属性。会话数据（本章后面将详细介绍）只是“记住”网站信息的一种方式。在下面几节中，我将介绍ASP.NET支持的几种主要方式。

源代码　SimpleStateExample网站的源代码位于Chapter 34子目录下。

34.2　ASP.NET状态管理技术

ASP.NET提供了几个可以用来在Web应用程序中维持状态信息的机制。确切地说，你有如下几个选项：

- 利用ASP.NET视图状态；
- 利用ASP.NET控件状态；
- 定义应用程序级变量；
- 利用缓存对象；
- 定义会话级变量；
- 定义cookie数据。

除了这些技术之外，ASP.NET还提供了现成的用户配置API来以永久的方式保存用户数据。我们会依次研究每一个方式，首先从ASP.NET视图状态开始。

34.3　ASP.NET视图状态的作用

此前术语视图状态（view state）已经使用了多次，但并未正式定义，所以这里我们来揭开这个术语的神秘面纱。如果没有框架的支持，在构建即将输出的HTTP响应时，Web开发者需要手动为传入的窗体部件重新填充值。

在ASP.NET下，我们不再需要手动删除和重新填充位于HTML部件内的值，因为ASP.NET运行时将自动嵌入一个隐藏的表单字段（名为__VIEWSTATE），其范围是浏览器和一个指定的页面之间。分配到这个字段的数据是一个64位编码的字符串，它包括一个描述当前页面上每个GUI部件值的名称/值对集。

System.Web.UI.Page基类的Init事件处理程序是一个实体，它负责读取在__VIEWSTATE字段内发现的传入值，以在派生类中填充适当的成员变量。（这就是为什么在一个页面的Init事件处理程序的作用域内访问Web部件的状态最冒险。）

同时，恰恰在输出响应被提交回发出请求的浏览器之前，__VIEWSTATE数据被用来重新填充窗体部件。显然，ASP.NET最大的优点就是不需要任何用户参与。当然，如果愿意的话，用户总是能够与默认功能交互，以及改变或者禁用默认功能。为了理解如何实现这些，我们来看一下具体的视图状态示例。

34.3.1 演示视图状态

首先，创建一个新的空网站应用程序，名为ViewStateApp并插入一个新的Web窗体。在初始的*.aspx页面上添加一个ASP.NET ListBox Web控件myListBox和一个Button类型控件btnPostback。

现在，使用Visual Studio Properties窗口，访问Items属性，并通过相关的对话框向ListBox添加4个ListItem。结果将为如下形式：

```
<asp:ListBox ID="myListBox" runat="server">
  <asp:ListItem>Item One</asp:ListItem>
  <asp:ListItem>Item Two</asp:ListItem>
  <asp:ListItem>Item Three</asp:ListItem>
  <asp:ListItem>Item Four</asp:ListItem>
</asp:ListBox>
```

注意，你是在*.aspx文件中直接硬编码了ListBox内的项。你已经知道，所有在ASP.NET Web Form内的<asp:>定义将在最后的HTTP响应前自动提交回它们的HTML代码（假如它们有runat="server"特性）。

<%@Page%>指令有一个可选的特性，名为EnableViewState，它在默认时设置为true。要禁用这个行为，只需更新<%@Page%>指令即可，如下所示：

```
<%@ Page Language="C#" AutoEventWireup="true"
        CodeFile="Default.aspx.cs" Inherits="_Default"
        EnableViewState ="false" %>
```

那么禁用状态视图的确切意义是什么呢？这要看情况。考虑先前术语的定义，你可能会认为，如果禁用*.aspx文件的视图状态，那么在到Web服务器的各次回传间，ListBox内的值将不再被保存。然而，如果要照现在这个样子运行这个应用程序，你也许会惊讶地发现，无论你回传到页面多少次，ListBox内的信息都被保留了。

事实上，如果检查返回到浏览器的源HTML（通过右击浏览器中的页面并选择View Source），你也许更惊讶地看到隐藏的__VIEWSTATE字段仍然存在：

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
 value="/wEPDwUKLTM4MTM2MDM4NGRkqGC6gjEV25JnddkJiRmoIc1OSIA=" />
```

34

但这里假设ListBox被动态填充在代码隐藏文件内，而不是HTML <form>定义内。首先，从当前的*.aspx文件中移除<asp:ListItem>声明，如下所示：

```
<asp:ListBox ID="myListBox" runat="server">
</asp:ListBox>
```

接着，填写代码隐藏文件中Load事件处理程序内的ListBox项：

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    // 动态填写ListBox
    myListBox.Items.Add("Item One");
    myListBox.Items.Add("Item Two");
    myListBox.Items.Add("Item Three");
    myListBox.Items.Add("Item Four");
  }
}
```

如果展示这个被更新的页面，你将发现，浏览器第一次请求页面时，ListBox内的值是存在的，并且是被占用的。然而，在回传后，ListBox内的值却突然没有了。ASP.NET视图状态的第一个规则是，只有在你拥有其值是通过代码动态生成的部件时，才能实现视图状态效果。如果在*.aspx文件的<form>标签内硬编码值，这些项的状态总是通过回传被保存（即使当为给定页面设置EnableViewState为false时）。

如果禁用全部的*.aspx文件的视图状态看起来有点太激进的话，回想一下，每个System.Web.UI.Control基类的派生类都继承了EnableViewState属性，这个属性使得一个控件接一个控件地禁用视图状态非常容易：

```
<asp:GridView id="myHugeDynamicallyFilledGridOfData" runat="server"
 EnableViewState="false">
</asp:GridView>
```

说明　从.NET 4.0开始，较大的视图状态数据将自动被压缩，以减少该隐藏表单字段的大小。

34.3.2　添加自定义视图状态数据

除了EnableViewState属性外，System.Web.UI.Control基类也提供了一个名为ViewState的继承属性。实际上，这个属性提供了对System.Web.UI.StateBag类型的访问，该类型描述了__VIEWSTATE字段内的所有数据。使用StateBag类型的索引器，能够使用一套名称/值对在隐藏的__VIEWSTATE表单字段内嵌入自定义信息。下面是一个简单的例子：

```
protected void btnAddToVS_Click(object sender, EventArgs e)
{
  ViewState["CustomViewStateItem"] = "Some user data";
  lblVSValue.Text = (string)ViewState["CustomViewStateItem"];
}
```

因为System.Web.UI.StateBag类型被设计用于操作System.Object类型的值，所以当希望访问一个给定键值时，你需要显式地把它转换为正确的数据类型（在这个例子中是System.String）。然而要注意的是，被放置在__VIEWSTATE字段内的值不能是任何字面上的对象。要特别注意的是，有效的类型只

有String、Integer、Boolean、ArrayList、HashTable或者这些类型的一个数组。

所以，由于*.aspx能够向__VIEWSTATE字符串插入一些自定义信息，所以下一步就是搞清楚什么时候需要这么做。大多数情况下，自定义视图状态数据对有特定需要的用户最适合。例如，你可以建立视图状态数据，它指定用户希望显示GridView（例如一个分类订单）的UI方式。视图状态数据并不太适用于保存完整的用户数据，例如购物车中的项目或缓存的DataSet。当你需要存储这个综合信息的类别时，必须使用会话数据或应用程序数据。在介绍它们之前，你需要理解Global.aspx文件的作用。

源代码 ViewStateApp网站的源代码位于Chapter 34子目录下。

34.4 Global.asax 文件的作用

到目前为止，ASP.NET应用程序可能看起来还是一堆*.aspx文件和它们各自的Web控件。虽然能够通过简单地链接一套相关的网页建立一个Web应用程序，但你可能更倾向于与作为整体的Web应用程序交互。为此，ASP.NET应用程序会通过Web Site→Add New Item菜单项选择包含一个可选的Global.asax文件，如图34-2所示（注意，你选择的是Global Application Class图标）。

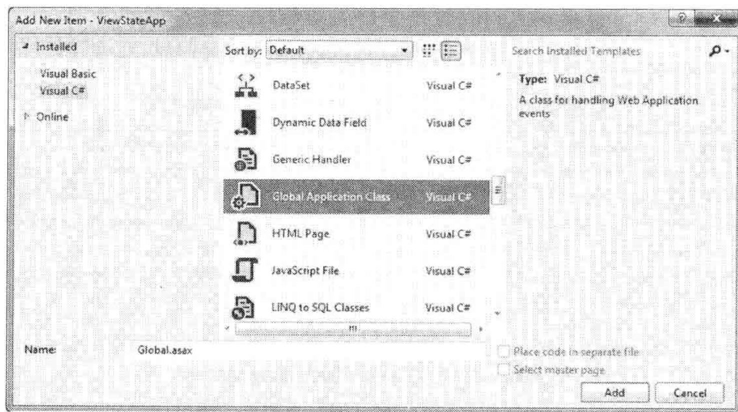


图34-2 Global.asax文件

简单地说，Global.asax与我们在ASP.NET中能够获得的传统的、可双击的*.exe很相近，这意味着这种类型表示站点自身的运行时行为。一旦将Global.asax文件插入到Web项目中，你将注意到它与一个包含一套事件处理程序的<script>块差不多，如下所示：

```
<%@ Application Language="C#" %>

<script runat="server">
  void Application_Start(object sender, EventArgs e)
  {
    // 运行应用程序启动的代码
  }
```

```
    void Application_End(object sender, EventArgs e)
    {
      // 运行应用程序关闭的代码
    }

    void Application_Error(object sender, EventArgs e)
    {
      // 当一个未处理过的错误发生时运行的代码
    }

    void Session_Start(object sender, EventArgs e)
    {
      // 当一个新的会话被启动时运行的代码
    }
    void Session_End(object sender, EventArgs e)
    {
      // 在会话结束的时候运行的代码
      // 说明：只有当web.config文件中的sessionstate模式设置为InProc的时候，
      // Session_End事件才会触发。如果会话模式设置为StateServer或SQLServer，事件就不会触发
    }
</script>
```

然而，外表是具有欺骗性的。在运行时，<script>块内的代码被装载进一个派生自System.Web HttpApplication的类类型。因此，在任何这些事件处理程序中，你都可以通过this或base关键字访问父类成员。

前面提到过，定义在Global.asax内的成员是事件处理程序，这些事件处理程序允许你与应用程序级（以及会话级）事件交互。表34-1列出了每个成员的作用。

表34-1　System.Web命名空间的核心类型

事件处理程序	作　用
Application_Start()	该事件处理程序在Web应用程序被启动后立刻被调用。因此，该事件在一个Web应用程序的整个生命周期内仅仅触发一次。这是定义Web应用程序全程所用到的应用程序级数据的理想位置
Application_End()	该事件处理程序在应用程序关闭时被调用。这会在最后一个用户超时或者通过IIS手动关闭了应用程序的情况下出现
Session_Start()	当新用户登录到你的应用程序时，该事件处理程序会触发。这里你可以确定回传期间你想保存的特定于用户的数据点
Session_End()	当用户的会话已经终止时，该事件处理程序会触发（通常情况下在预定义超时时发生）
Application_Error()	这是一个全局错误处理程序，在Web应用程序抛出一个未经处理的异常时被调用

34.4.1　全局最后异常事件处理程序

首先，介绍Application_Error()事件处理程序的作用。回想一下，特定的页面可以处理Error事件，以处理任何未处理的、出现在页面自身作用域的异常。同样，Application_Error()事件处理程序是处理未被指定页面处理的异常的最后关卡。对于页面级Error事件来说，你能够使用继承的Server属性访问指定的System.Exception，如下所示：

```
void Application_Error(object sender, EventArgs e)
{
  // 获得未处理的Error
  Exception ex = Server.GetLastError();

  // 在这里处理Error

  // 处理结束后消除Error
  Server.ClearError();
}
```

由于Application_Error()事件处理程序是Web应用程序处理异常最后的机会。通常我们会实现这个方法，把用户重定向到服务器上预定义的错误页面。这个方法另外一个常见的职责可能会包含发送电子邮件给Web管理员或写一个外部错误日志。

34.4.2　HttpApplication基类

上文已经提及Global.asax脚本动态地生成为一个从System.Web.HttpApplication基类派生的类，这个基类提供与System.Web.UI.Page类型同样的功能（用户界面不可见）。表34-2列出了其中的一些主要成员。

表34-2　System.Web.HttpApplication类型定义的主要成员

属　性	作　用
Application	该属性允许使用公开的HttpApplicationState类型与应用程序级变量进行交互
Request	该属性允许与传入的HTTP请求（通过HttpRequest底层对象）进行交互
Response	该属性允许与传入的HTTP响应（通过HttpResponse底层对象）进行交互
Server	该属性对当前的请求获取内在的服务器对象（通过HttpServerUtility底层对象）
Session	该属性允许使用HttpSessionState类型与会话级数据进行交互

同样，虽然Global.asax文件没有显式说明HttpApplication是底层的基类，但还是要记住所有“is-a”关系确实适用。

34.5　应用程序状态与会话状态的差别

在ASP.NET下，应用程序状态由HttpApplicationState类型的一个实例维护。这个类使你可以在所有登录到ASP.NET应用程序的用户（和所有页面）之间分享全局信息。不仅应用程序数据能被所有站点上的用户分享，而且如果一个用户改变了一个应用程序级数据点的值，在下一个回传中这个变化能够被所有其他用户看见。

另一方面，会话状态用来为一个指定用户（例如，购物车中的项）记忆成员信息。用户的会话状态物理上由HttpSessionState类类型描述。当一个新用户登录到一个ASP.NET Web应用程序时，运行库将自动分配给这个用户一个新的会话ID，默认状态下，它将在20分钟静止状态之后超时。这样，如果20 000个用户登录到了该站点，你就拥有20 000个不同的HttpSessionState对象，每个对象自动分配一个唯一的会话ID。Web应用程序和Web会话之间的关系如图34-3所示。

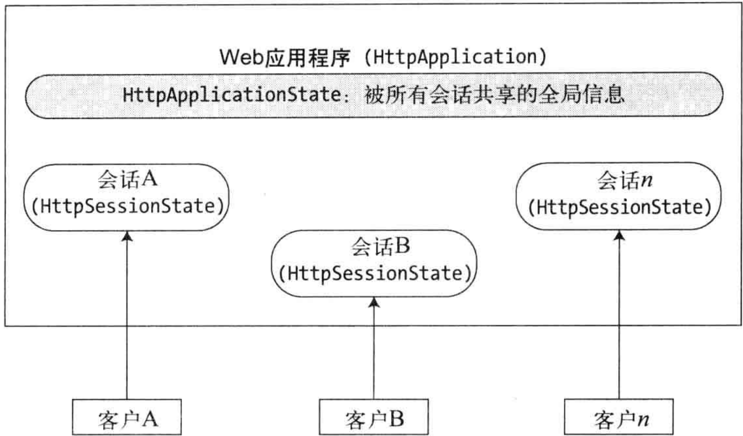


图34-3　应用程序状态与会话状态的差别

34.5.1　维护应用程序级的状态数据

HttpApplicationState类型允许开发人员在一个ASP.NET应用程序中的多个会话之间分享全局信息。表34-3描述了这个类型的一些核心成员。

表34-3　HttpApplicationState类型的成员

成　员	作　用
Add()	该方法允许为HttpApplicationState对象增加一个新的名称/值对。注意，这个方法通常不方便HttpApplicationState类的索引器使用
AllKeys	该属性返回string对象的一个数组，该string对象表示HttpApplicationState类型里的所有名字
Clear()	该方法删除HttpApplicationState类型里的所有项。在功能上这相当于RemoveAll()方法
Count	该属性获取HttpApplicationState类型里对象的数目
Lock()和Unlock()	这两个方法用于当希望以线程安全的方式修改一套应用程序变量时
RemoveAll()、Remove()和RemoveAt()	这些方法去除HttpApplicationState对象内一个特定的项（通过字符串名字）。RemoveAt()通过一个数值索引器去除项

为了演示如何使用应用程序状态创建一个新的空网站项目AppState，并插入一个新的Web窗体。然后插入一个新的Global.asax文件。在创建可以被所有用户共享的数据成员时，需要建立一套名称/值对。大多数情况下，最好在Global.asax.cs中的Application_Start()事件处理程序内建立这些成员，例如：

```
void Application_Start(Object sender, EventArgs e)
{
  // 设置一些应用程序变量
  Application["SalesPersonOfTheMonth"] = "Chucky";
  Application["CurrentCarOnSale"] = "Colt";
```

```
    Application["MostPopularColorOnLot"] = "Black";
  }
```

在Web应用程序的生命周期内（也就是说，直到Web应用程序被手动关闭或直到最后一个用户超时），如果需要的话，任何用户（在任何页面上）都可以访问这些值。假设你拥有一个页面，它将通过一个按钮的Click事件处理程序展现Label内当前的汽车折扣，如下所示：

```
protected void btnShowCarOnSale_Click(object sender, EventArgs arg)
{
  lblCurrCarOnSale.Text = string.Format("Sale on {0}'s today!",
    (string)Application["CurrentCarOnSale"]);
}
```

与ViewState属性类似，注意如何必须将HttpApplicationState对象返回的值转换为正确的底层类型，因为Application属性在一般的System.Object类型上操作。

现在，假设Application属性能够保存任何类型，那么显然能够在站点的应用程序状态内放置自定义类型（或任何.NET对象）。假设你希望使用一个名为CarLotInfo的强类型对象来维护当前的3个应用程序变量，如下所示：

```
public class CarLotInfo
{
  public CarLotInfo(string s, string c, string m)
  {
    salesPersonOfTheMonth = s;
    currentCarOnSale = c;
    mostPopularColorOnLot = m;
  }
  public string SalesPersonOfTheMonth { get; set; };
  public string currentCarOnSale { get; set; };
  public string mostPopularColorOnLot { get; set; };
}
```

利用这个辅助类，可以修改Application_Start()事件处理程序，如下所示：

```
void Application_Start(Object sender, EventArgs e)
{
  // 在应用程序数据段中放置一个自定义的对象
  Application["CarSiteInfo"] =
    new CarLotInfo("Chucky", "Colt", "Black");
}
```

然后在一个Button控件（btnShowAppVariables）的服务器端Click事件处理程序内使用公开字段数据访问信息，如下所示：

```
protected void btnShowAppVariables_Click(object sender, EventArgs e)
{
  CarLotInfo appVars =
    ((CarLotInfo)Application["CarSiteInfo"]);
  string appState =
    string.Format("<li>Car on sale: {0}</li>",
    appVars.CurrentCarOnSale);
  appState +=
    string.Format("<li>Most popular color: {0}</li>",
    appVars.MostPopularColorOnLot);
  appState +=
    string.Format("<li>Big shot SalesPerson: {0}</li>",
    appVars.SalesPersonOfTheMonth);
```

```
  lblAppVariables.Text = appState;
}
```

由于当前要销售的汽车数据由自定义的类类型表示，我们的btnShowCarOnSalieSalie_Click Click事件处理程序还需要如下进行更新：

```
protected void btnShowCarOnSale_Click(object sender, EventArgs e)
{
  lblCurrCarOnSale.Text = String.Format("Sale on {0}'s today!",
    ((CarLotInfo)Application["CarSiteInfo"]).CurrentCarOnSale);
}
```

34.5.2　修改应用程序数据

在Web应用程序执行期间，可以使用HttpApplicationState类型的成员通过编程更新或删除任何或全部应用程序级数据项。例如，要删除一个指定项，只需要调用Remove()方法即可。如果你希望销毁所有应用程序级数据，调用RemoveAll()：

```
private void CleanAppData()
{
  // 通过字符串名字删除一个项
  Application.Remove("SomeItemIDontNeed");

  // 销毁全部应用程序数据
  Application.RemoveAll();
}
```

如果你只希望改变一个已存在的应用程序级数据项的值，就只需要对相应的数据项进行新的赋值。假设页面现在支持一个新的Button类型，该类型允许用户通过读入名为txtNewSP的TextBox的值来用户改变当前的最佳销售员。Click事件处理程序如下：

```
protected void btnSetNewSP_Click(object sender, EventArgs e)
{
  // 设置新的销售员
  ((CarLotInfo)Application["CarSiteInfo"]).SalesPersonOfTheMonth
    = txtNewSP.Text;
}
```

如果运行这个Web应用程序，你将发现这个应用程序级数据项已经被更新了。此外，由于应用程序变量对于所有的用户会话来说都是可访问的，如果要登录3个或4个Web浏览器实例，那么当一个实例改变了当前的销售员时，其他的每个浏览器都会在回传时显示新值。图34-4显示了可能的输出结果。

需要理解一点：如果出现一组应用程序级变量必须作为一个单元更新的情况，你可能会面临数据损坏的风险（当另一个用户企图访问一个应用程序级数据点时，从技术上讲，它可能被改变）。你能够使用System.Threading命名空间的线程手动锁住逻辑，但这个HttpApplicationState类型有如下的两个方法——Lock()和Unlock()，它们能够自动确保线程安全性：

```
// 安全访问相关应用程序数据
Application.Lock();
Application["SalesPersonOfTheMonth"] = "Maxine";
Application["CurrentBonusedEmployee"] = Application["SalesPersonOfTheMonth"];
Application.UnLock();
```

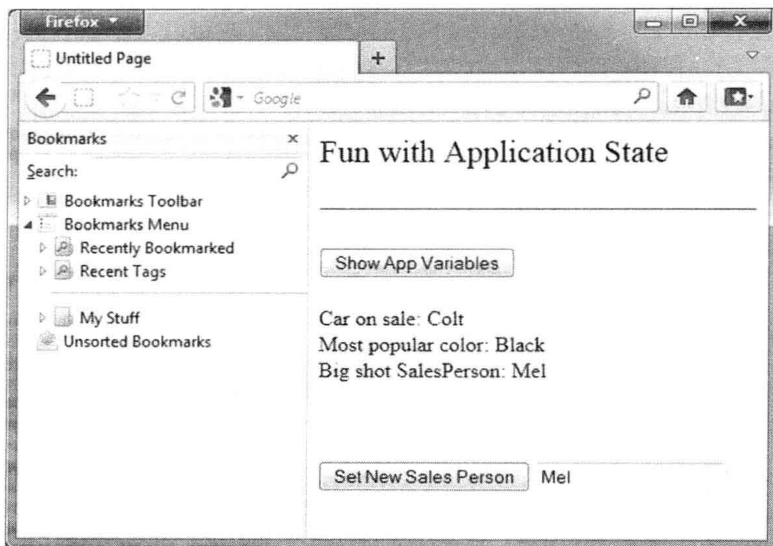


图34-4 显示应用程序数据

34.5.3 处理Web应用程序的关闭

HttpApplicationState类型可维护其所包含项的值，直到下列一种情形发生：最新登录到站点的用户超时（或手动退出）或者某人通过IIS手动关闭了站点。在每种情况中，都将自动调用派生自HttpApplication类型的Application_End()方法。在这个事件处理程序中，你能够执行任何种类的必要的清理代码：

```
void Application_End(Object sender, EventArgs e)
{
  // 向一个数据库或其他任何需要的位置写当前应用程序变量
}
```

源代码 AppState网站的源代码位于Chapter 34子目录下。

34.6 使用应用程序缓存

ASP.NET提供了另一个更灵活的处理应用程序级数据的方式。如前文所述，HttpApplicationState对象内的值在Web应用程序处于活动状态并且正被访问时一直驻留在内存中。然而有时，你可能只希望在某个时段维护一些应用程序数据。例如，你可能希望获得一个只在5分钟内有效的ADO.NET DataSet。在这段时间过后，你可能想要获得一个最新的DataSet来获知可能的数据库更改。虽然从技术上讲，使用HttpApplicationState和某种手工监视器创建这个结构是可行的，但是使用ASP.NET应用程序缓存可以使你的工作变得简单。

顾名思义，这个ASP.NET System.Web.Caching.Cache对象（它可以通过Context.Cache属性访问）允许你定义一个在设定时间内可以被所有用户（从所有页面）访问的对象。在最简单的情况下，与缓存交互和与HttpApplicationState类型交互是一样的：

```
// 向缓存添加一个项
// 这个项不会失效
Context.Cache["SomeStringItem"] = "This is the string item";

// 从缓存中获得项
string s = (string)Context.Cache["SomeStringItem"];
```

说明　如果你希望在Global.asax内访问Cache，需要使用Context属性。尽管如此，如果你正位于一个派生自System.Web.UI.Page的类型作用域内，就能够通过页面的Cache属性直接访问Cache对象。

在类型索引器的基础上，System.Web.Caching.Cache类只定义了很少数量的成员。例如，Add()方法用来将一个当前未定义的新项插入缓存（如果这个指定项已经存在了，Add()就什么也不做）。Insert()方法也将向缓存中放置一个成员。然而，如果该项在当前已定义了，Insert()将用一个新的对象替换当前项。因为这是经常发生的行为，我将专门研究Insert()方法。

34.6.1　使用数据缓存

让我们看一个例子。首先，创建一个新的空网站项目CacheState，然后插入一个Web窗体和一个Global.asax文件。就像一个被HttpApplicationState类型维护的应用程序级数据项一样，这个Cache可以保存任何派生自System.Object的类型，并且能够频繁地在Application_Start()事件处理程序内被填充。这个例子的目的是每15秒自动更新一次某个DataSet的内容。这个DataSet将包含一组我们讨论ADO.NET时创建的AutoLot数据库的Inventory表的记录。

现在，设置对AutoLotDAL.dll的引用（见第21章），并更新Global.asax文件，如下所示（随后分析代码）：

```
<%@ Application Language="C#" %>
<%@ Import Namespace = "AutoLotConnectedLayer" %>
<%@ Import Namespace = "System.Data" %>

<script runat="server">
  // 定义一个静态级的Cache成员变量
  static Cache theCache;

  void Application_Start(Object sender, EventArgs e)
  {
    // 首先分配静态'theCache'变量
    theCache = Context.Cache;

    // 当启动应用程序时，在AutoLot数据库的Inventory表中读取当前记录
    InventoryDAL  dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
      "Initial Catalog=AutoLot;Integrated Security=True");
    DataTable theCars = dal.GetAllInventoryAsDataTable();
    dal.CloseConnection();
```

```
    // 现在，在缓存中存储DataSet
    theCache.Insert("AppDataTable",
      theCars,
      null,
      DateTime.Now.AddSeconds(15),
      Cache.NoSlidingExpiration,
      CacheItemPriority.Default,
      new CacheItemRemovedCallback(UpdateCarInventory));
  }

  // CacheItemRemovedCallback委托的目标
  static void UpdateCarInventory(string key, object item,
    CacheItemRemovedReason reason)
  {
    InventoryDAL  dal = new InventoryDAL();
    dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
      "Initial Catalog=AutoLot;Integrated Security=True");
    DataTable theCars = dal.GetAllInventoryAsDataTable();
    dal.CloseConnection();

    // 现在存储在缓存中
    theCache.Insert("AppDataTable",
      theCars,
      null,
      DateTime.Now.AddSeconds(15),
      Cache.NoSlidingExpiration,
      CacheItemPriority.Default,
      new CacheItemRemovedCallback(UpdateCarInventory));
  }
...
</script>
```

首先，注意我们定义了一个静态的Cache成员变量。理由是，你也定义了两个需要访问Cache对象的静态成员。回想一下，静态方法不访问继承的成员，因此你不能使用Context属性。

在Application_Start()事件处理程序中，填充一个DataTable，然后把对象放置在应用程序缓存内。你会想到，Context.Cache.Insert()方法已经重载了多次。这里，为每个可能的参数提供一个值。看看下面被注释掉的对Insert()的调用：

```
theCache.Insert("AppDataTable",   // 用来识别缓存中各项的名字
  theCars,                        // 放置在缓存中的对象
  null,                           // 这个对象的任何依赖
  DateTime.Now.AddSeconds(15),    // 绝对超时值
  Cache.NoSlidingExpiration,      // 不要使用滑动期限（如下）
  CacheItemPriority.Default,      // 缓存项的权限级别
  // 用于CacheItemRemove事件的委托
  new CacheItemRemovedCallback(UpdateCarInventory));
```

前两个参数简单编辑了项的名称/值对，第三个参数允许定义一个CacheDependency对象（在这个例子里，它是空的，因为DataTable不依赖任何东西）。

参数DateTime.Now.AddSeconds(15)指定了绝对过期时间。这意味着缓存项将在15秒之后从缓存中被完全清除。绝对过期对于那些需要经常刷新的数据项（如股票行情）来说，是十分有用的。

Cache.NoSlidingExpiration参数指定了缓存项不使用可调过期。可调过期可以使缓存中的项至少保存一个固定的时间。例如，设置某个缓存项的可调过期为60 s，那么它可以在缓存中至少存活60 s。如果任何Web页面在这个时间内访问该缓存项，时钟将被重置，该缓存项可以再存活60 s。如果60 s

内没有Web页面访问该项，将从缓存中移除它。可调过期对于那些生成代价高（时间上）但并不被Web页面频繁使用的数据来说，是十分有用的。

注意，你不能同时为给定的缓存项指定绝对过期和可调过期。设置绝对过期时，要使用Cache.NoSlidingExpiration；设置可调到期时，要使用Cache.NoAbsoluteExpiration。最后，正如UpdateCarInventory()方法的签名所示，CacheItemRemovedCallback委托只能调用匹配如下签名的方法：

```
void UpdateCarInventory(string key, object item, CacheItemRemovedReason reason)
{
}
```

至此，当启动应用程序时，DataTable就被填充和缓存了。每隔15秒，DataTable就被清除、更新，然后再插入到缓存中。为了看到这么做的结果，需要创建一个允许一定程度的用户交互的Page。

34.6.2　修改*.aspx文件

在图34-5中可以看到，我构建了一个允许用户输入必要数据来向数据库插入新记录的UI（通过4个TextBox控件）。这一个Button控件的Click事件将处理数据库操作。最后，除了一些描述性的Label控件外，页面底部的GridView将用来显示Inventory表中的当前记录。

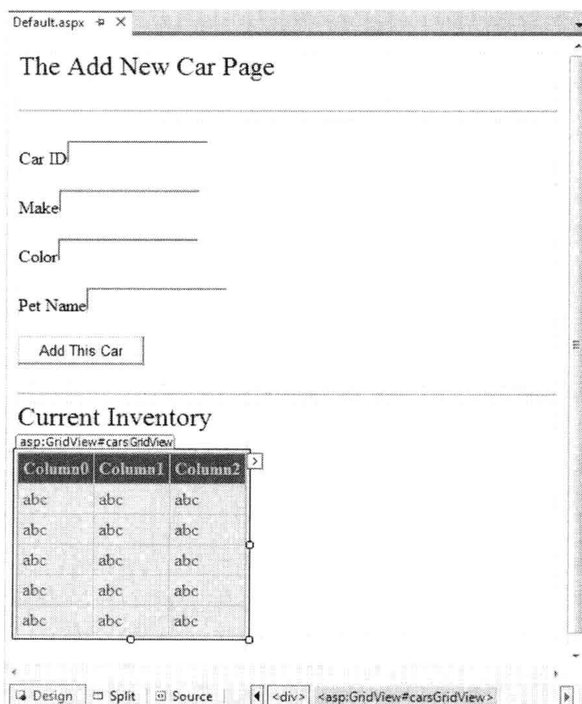


图34-5　缓存应用程序GUI

在页面的Load事件处理程序中，配置GridView以显示用户首次访问页面时缓存的DataTable的内容（确保在代码文件中导入了System.Data和AutoLotConnectedLayer命名空间）：

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!IsPostBack)
  {
    carsGridView.DataSource = (DataTable)Cache["AppDataTable"];
    carsGridView.DataBind();
  }
}
```

在Add This Car按钮的Click事件处理程序里，使用InventoryDAL类型向AutoLot数据库中插入新的记录。一旦插入了记录，调用名为RefreshGrid()的辅助函数，它将更新UI：

```
protected void btnAddCar_Click(object sender, EventArgs e)
{
  // 更新Inventory表，调用RefreshGrid()
  InventoryDAL dal = new InventoryDAL();
  dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
    "Initial Catalog=AutoLot;Integrated Security=True");
  dal.InsertAuto(int.Parse(txtCarID.Text), txtCarColor.Text,
    txtCarMake.Text, txtCarPetName.Text);
  dal.CloseConnection();
  RefreshGrid();
}

private void RefreshGrid()
{
  InventoryDAL dal = new InventoryDAL();
  dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;" +
    "Initial Catalog=AutoLot;Integrated Security=True");
  DataTable theCars = dal.GetAllInventoryAsDataTable();
  dal.CloseConnection();

  carsGridView.DataSource = theCars;
  carsGridView.DataBind();
}
```

现在，为了测试缓存的使用，首先运行当前的程序（Ctrl+F5）并且将浏览器中的URL复制到剪贴板上。然后，运行第二个IE浏览器的实例（使用Start按钮）并且将URL粘贴到这个实例上。现在我们应该有两个Web浏览器的实例，都查看了Default.aspx并显示了相同的数据。

在浏览器的一个实例中，新增一个汽车条目。很明显，可以从更新后的GridView上看到结果，并且浏览器也回发了。

在第二个浏览器实例中，单击Refresh按钮（F5）。看不见新的项，因为Page_Load事件处理程序正在直接从缓存中读取。（如果你确实看见了这个值，那么15秒已经到期了。敲击快一点或增加时长都可以使DataTable驻留在缓存中。）等待几秒，然后从第二个浏览器实例中再次单击Refresh按钮。现在，你将看到新的项了，因为缓存内的DataTable已经过期并且CacheItemRemovedCallback委托目标方法已经自动更新了缓存的DataTable。

可见，Cache类型的主要优点是，它能确保当一个成员被删除后有机会做出反应。在这个例子中，你当然能够避免使用Cache，只是让Page_Load()事件处理程序始终直接从AutoLot数据库中读取数据（但这比缓存方法慢）。不过，有一点应该清楚：缓存允许使用.NET委托自动刷新数据。

源代码　CacheState网站的源代码位于Chapter 34子目录下。

34

34.7　维护会话数据

对应用程序级数据和缓存数据的介绍就到这里。接下来，我们讨论按用户存储数据的作用。像先前提过的，会话（session）就是给定用户与Web应用程序之间的交互，会话由HttpSessionState对象表示。为了维护一个特定用户的状态信息，可以使用网页类或Global.asax中的Session属性。在线购物车是必须按用户维护数据的典型例子。如果10个人全部登录到一个在线商场，每个个体都将有一组她（他）想要购买的项并且这个数据需维护。

当新的用户登录到Web应用程序时，.NET运行库将自动给这个用户分配一个唯一的会话ID，用来识别这个用户。每个会话ID被分配一个自定义的HttpSessionState类型实例，以保存该用户的数据。插入或获取会话数据与操作应用程序数据从语法上来说是一样的，例如：

```
// 为当前用户添加/获取一个会话
Session["DesiredCarColor"] = "Green";
string color = (string) Session["DesiredCarColor"];
```

在Global.asax中，允许通过Session_Start()和Session_End()事件处理程序来拦截会话的起始点和结束点。通过Session_Start()，你可以随意创建用户的数据项，而Session_End()允许在用户会话结束时执行任何想执行的任务：

```
<%@ Application Language="C#" %>
...
void Session_Start(Object sender, EventArgs e)
{
  // 新会话。如果需要，就准备好
}
void Session_End(Object sender, EventArgs e)
{
  // 用户登出/超时。如果需要，就销毁
}
```

就像应用程序状态一样，会话状态会保存任何派生自System.Object的类型，包括自定义类。例如，假设你拥有一个新的空网站项目（名为SessionState），它定义了名为UserShoppingCart的辅助类，如下所示：

```
public class UserShoppingCart
{
  public string DesiredCar { get; set; };
  public string DesiredCarColor { get; set; };
  public float DownPayment { get; set; };
  public bool IsLeasing { get; set; };
  public DateTime DateOfPickUp { get; set; };

  public override string ToString()
  {
    return string.Format
     ("Car: {0}<br>Color: {1}<br>$ Down: {2}<br>Lease: {3}<br>Pick-up Date: {4}",
       DesiredCar, DesiredCarColor, DownPayment, IsLeasing,
       DateOfPickUp.ToShortDateString());
  }
}
```

现在，插入Global.asax文件。在这个Session_Start()事件处理程序中，你现在能够分配给每个用户一个新的UserShoppingCart类的实例，如下所示：

```
void Session_Start(Object sender, EventArgs e)
{
  Session["UserShoppingCartInfo"]= new UserShoppingCart();
}
```

当用户浏览完所有Web页面时，你可以获取UserShoppingCart实例，然后用针对每个用户的具体数据填充该实例的字段。例如，假设有一个简单的*.aspx页面，它定义了一组输入控件，分别对应于UserShoppingCart类型的每个字段、一个用于设置值的Button和两个用来显示用户会话ID和会话信息的Label（如图34-6所示）。



图34-6 会话应用程序GUI

Button控件的服务器端Click事件处理程序非常简单（从TextBox中把值取出，然后在一个Label控件上显示购物车数据）：

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
  // 设置当前用户设定
  UserShoppingCart cart =
    (UserShoppingCart)Session["UserShoppingCartInfo"];
  cart.dateOfPickUp = myCalendar.SelectedDate;
  cart.DesiredCar = txtCarMake.Text;
  cart.DesiredCarColor = txtCarColor.Text;
```

```
        cart.DownPayment = float.Parse(txtDownPayment.Text);
        cart.IsLeasing = chkIsLeasing.Checked;
        lblUserInfo.Text = cart.ToString();
        Session["UserShoppingCartInfo"] = cart;
    }
```

在Session_End()中，你可能希望把UserShoppingCart字段的值持久保存在数据库或者其他地方。（然而，在本章结束的部分我们会看到，ASP.NET用户配置API会自动完成这些。）同样，我们可能希望实现Session_Error()来捕获任何错误输入（或者可以在Default.aspx页面中使用各种验证控件来处理这样的用户错误）。

不管怎么样，如果启动两个或三个你喜欢的浏览器实例指向相同URL（像数据缓存示例一样进行复制、粘贴操作），你就会发现每个用户都可以构建映射到其唯一的HttpSessionState实例的购物车。

HttpSessionState的其他成员

HttpSessionState类在类型索引器上定义了一些其他成员。首先，SessionID属性将返回当前用户的唯一ID。如果要查看本例中自动赋值的会话ID，可以像下面这样处理页面的Load事件：

```
protected void Page_Load(object sender, EventArgs e)
{
  lblUserID.Text = string.Format("Here is your ID: {0}",
    Session.SessionID);
}
```

Remove()和RemoveAll()方法可以用来清除用户的HttpSessionState实例的项，例如：

```
Session.Remove("SomeItemWeDontNeedAnymore");
```

HttpSessionState类型也定义了一组控制当前会话的过期策略的成员。在HttpSessionState对象销毁前，假定默认每个用户拥有20分钟的静止状态。这样，如果一个用户进入Web应用程序（因此获得一个唯一的会话ID），但是没有在20分钟内返回到站点，运行库将假设用户不再感兴趣并且销毁那个用户的所有会话数据。使用Timeout属性可以逐个用户地改变这个默认的20分钟过期时间。这个设置通常位于Session_Start()方法的作用域内，如下所示：

```
void Session_Start(Object sender, EventArgs e)
{
  // 每个用户有5分钟的静止时间
  Session.Timeout = 5;
  Session["UserShoppingCartInfo"]
    = new UserShoppingCart();
}
```

说明　如果你不需要改变每个用户的Timeout值，可以通过web.config文件内<sessionState>元素的timeout特性为所有用户改变默认的20分钟设置（详见本章末尾）。

Timeout属性的优点是你可以为每个用户分别分配特定的超时值。例如，想象你已经创建了一个Web应用程序，它允许用户支付现金获得一定级别的会员资格。你可能说金卡成员应该允许一个小时的超时，但普通成员应该只允许30秒。这个假设遇到了问题，如何在Web访问那边保存用户特定的信

息（例如当前会员级别）？一个可能的答案是通过HttpCookie类型的用户。（这里谈到了cookie……）

源代码 SessionState网站的源代码位于Chapter 34子目录下。

34.8 cookie

这里研究的最后一个状态管理技术是在cookie内保持数据的动作，它总是在用户的计算机上呈现为一个文本文件（或文件集）。当用户登录到一个给定站点时，浏览器检查用户的机器是否有一个相应的URL的cookie文件，如果有，会将这个数据追加到HTTP请求中。

然后负责接收的服务器端网页就能够读取cookie数据，创建一个可能适合当前用户偏好的GUI。相信你已经注意到，当你访问最喜欢的站点时，它竟然知道你希望看到的内容。（部分）原因是由于存储在你的电脑中的cookie，其中包含了指定网站的信息。

说明 cookie文件的确切位置取决于所使用的浏览器和操作系统。

给定cookie文件的内容将由于URL的不同而发生明显变化，但是谨记：它们最终都是文本文件。因此，当你希望维护关于当前用户的敏感信息时（例如信用卡号、密码等），cookie是一个可怕的选择。即使你花时间加密数据，狡猾的黑客仍能够破解这些值并且恶意利用。但无论如何，cookie在Web应用程序的开发中都扮演着重要角色，所以下面研究ASP.NET如何处理这个特殊的状态管理技术。

34.8.1 创建cookie

首先理解一点，ASP.NET cookie既可以配置成永久的也可以配置成临时的。一个持久的cookie通常当成是cookie数据的传统定义，因为名称/值对集在物理上保存在用户的硬盘。临时的cookie（术语也称为会话cookie）与持久的cookie包含同样的数据，但是名称/值对从不保存在用户的硬盘中，它们仅存在于HTTP首部里。一旦用户关闭浏览器，包含在会话cookie内的所有数据都被销毁。

System.Web.HttpCookie类型是表示（永久的和临时的）cookie数据的服务器端的类。当希望在网页代码中创建一个新的cookie时，可以访问Response.Cookies属性。一旦这个新的HttpCookie插入到内部集合内，名称/值对就跟随HTTP首部返回到浏览器。

为了直接查看cookie行为，创建一个新的空网站项目（CookieStateApp），然后如创建如图34-7所示的第一个Web窗体（需要你插入）的UI。

在第一个按钮的Click事件处理程序内，创建一个新的HttpCookie，并且把它插入到由HttpRequest.Cookies属性提供的Cookie集合里。注意，数据将不持久在用户的硬盘里，除非你明确使用HttpCookie.Expires属性设置一个到期日。这样，下列代码将创建一个临时的cookie，当用户关闭浏览器时被销毁：

```
protected void btnCookie_Click(object sender, EventArgs e)
{
  // 创建一个新的（临时的）cookie
```

```
    HttpCookie theCookie =
      new HttpCookie(txtCookieName.Text,
      txtCookieValue.Text);
    Response.Cookies.Add(theCookie);
}
```

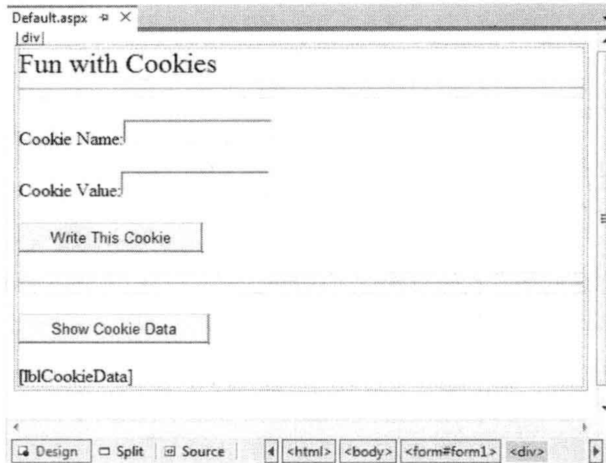


图34-7　CookieStateApp的UI

然而下列代码将生成一个从今天起3个月后过期的持久性cookie：

```
protected void btnCookie_Click(object sender, EventArgs e)
{
    HttpCookie theCookie =
      new HttpCookie(txtCookieName.Text,
      txtCookieValue.Text);
    theCookie.Expires = DateTime.Now.AddMonths(3);
    Response.Cookies.Add(theCookie);
}
```

34.8.2　读取传入的cookie数据

回想一下，当导航到一个以前访问过的页面时，浏览器是负责访问持久化cookie的实体。如果浏览器决定向服务器发送cookie,需要通过HttpRequest.Cookies属性访问*.aspx页面中传入的数据。举例说明，实现第二个按钮的Click事件处理程序：

```
protected void btnShowCookie_Click(object sender, EventArgs e)
{
    string cookieData = "";
    foreach (string s in Request.Cookies)
    {
      cookieData +=
       string.Format("<li><b>Name</b>: {0}, <b>Value</b>: {1}</li>",
       s, Request.Cookies[s].Value);
    }
    lblCookieData.Text = cookieData;
}
```

如果现在运行这个应用程序并且单击新的按钮，你将发现cookie数据已经真正地被浏览器发送了并且可以在服务器端*.aspx代码中成功访问。

源代码 CookieStateApp网站的源代码位于Chapter 34子目录下。

34.9 <sessionState>元素的作用

至此，我们已经研究了各种记住用户信息的方式。我们已经看到了，维护视图状态和应用程序、缓存、会话以及cookie数据的方式（通过类索引器）都差不多。我们还看到了，Global.asax的方法通常用于在Web应用程序生命周期中对事件进行拦截和响应。

默认情况下，ASP.NET会存储进程中的会话状态。好处是可以尽快访问信息。然而，缺点是如果应用程序域崩溃了（不管什么原因），用户的所有状态数据就没有了。此外，如果我们以进程中*.dll方式保存数据的话，就不能和联网Web场进行交互。如果我们的Web应用程序由一个Web服务器承载的话，这个默认的存储方式就可以。然而，你可能会想，这个模型对于Web服务器场来说不是很理想，因为会话状态会仅仅局限在某个应用程序域中。

34.9.1 在ASP.NET会话状态服务器中保存会话数据

在ASP.NET下，我们可以让运行库在叫做ASP.NET会话状态服务器（aspnet_state.exe）的辅助进程中承载会话状态*.dll。这样的话，我们就可以把*.dll从aspnet_wp.exe拆分到单独的*.exe中，它可以位于Web场的任何机器上。即使我们希望在和Web服务器相同的机器上运行aspnet_state.exe进程，我们也可以获得把状态数据划分在单独进程中的优势（因为这样能更持久）。

要使用会话状态服务器，第一步就是启动目标机器上的aspnet_state.exe Windows服务。只要在Developer Command Prompt窗口中输入下面的代码就可以了：

```
net start aspnet_state
```

或者，我们可以从控制面板中的管理工具文件夹使用服务小程序来启动aspnet_state.exe，如图34-8所示。

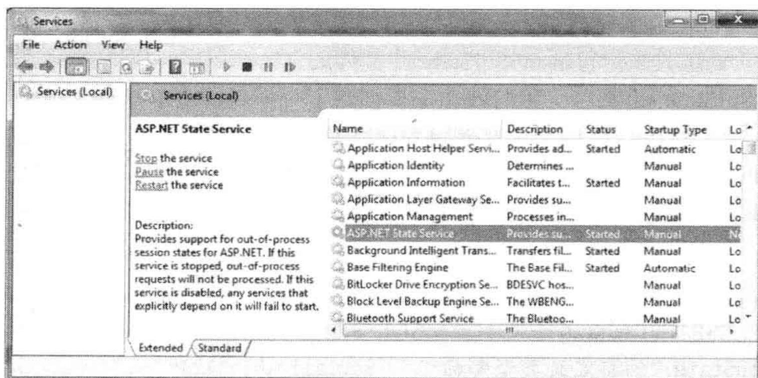


图34-8 使用服务小程序开启aspnet_state.exe

这个方式最主要的优势是，我们可以使用Properties窗口配置aspnet_state.exe，让它在机器启动的时候自动启动。不管怎么样，只要会话服务器在运行，就需要按如下所示为web.config文件增加<sessionState>元素：

```
<system.web>
  <sessionState
    mode="StateServer"
    stateConnectionString="tcpip=127.0.0.1:42626"
    sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
    cookieless="false"
    timeout="20"
  />
...
</system.web>
```

就这么简单！至此，CLR就可以在aspnet_state.exe中承载会话相关的数据了。这样，如果承载Web应用程序的应用程序域崩溃了，会话状态就会保留下来。还要注意，<sessionState>元素还可以支持stateConnectionString特性。默认的TCP/IP地址值（127.0.0.1）指向本机。如果希望让.NET运行库使用另一个联网机器上的aspnet_state.exe服务（同样，考虑一下Web场），完全可以更新这个值。

34.9.2　把会话数据保存在专门的数据库中

最后，如果需要Web应用程序具有高度的隔离性和持久性，可以让运行库把所有的会话状态数据都保存在微软SQL Server中。对于web.config的更新相当简单：

```
<sessionState
  mode="SQLServer"
  stateConnectionString="tcpip=127.0.0.1:42626"
  sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
  cookieless="false"
  timeout="20"
/>
```

然而，在尝试运行相关的Web应用程序之前，我们需要确保目标机器（由sqlConnectionString特性指定）已经进行了正确的配置。安装.NET Framework 4.5 SDK（或适当的Visual Studio）的时候，我们会得到两个文件InstallSqlState.sql和UninstallSqlState.sql，它们默认位于C:\Windows\Microsoft.NET\Framework\<*version*>下。在目标机器上，我们必须使用诸如微软SQL Server Management Studio（微软SQL Server自带的）的工具来运行InstallSqlState.sql文件。

在运行InstallSqlState.sqt之后，我们就会发现创建了一个新的SQL Server数据库（ASPState），它包含了许多由ASP.NET运行库调用的存储过程以及一组用于存储会话数据本身的表（同样，为了交换数据，tempdb数据库也新增了一组表）。你可能会猜想，配置Web应用程序把会话数据保存在SQL Server中是所有选项中最慢的一种。但是其优势是用户数据是非常持久的（即使Web服务器重新启动）。

说明　如果你使用ASP.NET会话状态服务器或SQL Server来保存会话数据的话，必须确保任何放在HttpSessionState中的自定义类型都标记了[Serializable]特性。

34.10 ASP.NET 用户配置 API

到这里，我们已经研究了许多能让我们记住用户级别或应用程序级别数据的技术。但是，许多网站需要具备跨会话持久用户信息的能力。例如，可能需要为用户提供在网站上构建账号的能力，需要跨会话持久化ShoppingCart类的实例（对于网上商店网站）也可能希望持久化基本的用户喜好（主题等）。

虽然我们可以构建自定义的数据库（以及一些存储过程）来保存此类信息，而且还需要构建自定义代码库来和这些数据库对象进行交互。这不是一个复杂的任务，但是不管怎么样我们需要自己实现这样的基础设施。

为了简化这个事情，ASP.NET自带了现成的用户配置管理API以及数据库系统来解决这个问题。除了提供必要的基础设施，用户配置API还允许我们直接在web.config文件中定义要保存的数据（为了简单）。然而，我们也可以持久保存任何[Serializable]类型。在我们进行深入研究之前先来看看用户配置API会把指定数据保存在哪里。

34.10.1 ASPNETDB.mdf数据库

每一个使用Visual Studio构建的ASP.NET网站都自动提供了一个App_Data子目录。默认情况下，用户配置API（以及其他诸如ASP.NET角色成员管理API等服务）被配置为使用本地的叫ASPNETDB.mdf的SQL Server数据库，这个数据库位于App_Data文件夹中。这个默认行为在当前机器.NET安装目录中的machine.config文件中进行配置。其实，如果代码使用任何需要App_Data文件夹的ASP.NET服务的话，ASPNETDB.mdf数据文件都会在不存在副本的时候自动创建。

如果你更希望ASP.NET运行库和位于其他网络计算机的ASPNETDB.mdf进行通信，或者希望在SQL Server 7.0（或更高版本）实例上安装这个数据库，就需要使用aspnet_regsql.exe命令行工具来手动构建ASPNETDB.mdf。和其他好的命令行工具相似，aspnet_regsql.exe提供了许多选项，然而，如果我们按如下所示不带参数运行这个工具（通过Developer Command Prompt窗口）的话：

```
aspnet_regsql
```

就会启动一个基于GUI的向导来帮助我们在所选机器上创建安装SQL Server版本的ASPNETDB.mdf。

现在，假设我们的网站没有使用App_Data文件夹下数据库的本地副本，最后一步就是更新web.config文件来指出ASPNETDB.mdf的确切位置。假设我们在一个叫ProductionServer的机器上安装ASPNETDB.mdf。下面的（部分）web.config文件可以用来告诉用户配置API在默认位置可以找到必要的数据库项（你可以添加自定义的web.config来更改默认设置）：

```
<configuration>
  <connectionStrings>
    <add name="LocalSqlServer"
         connectionString ="Data Source=ProductionServer;Integrated
         Security=SSPI;Initial Catalog=aspnetdb;"
         providerName="System.Data.SqlClient"/>
  </connectionStrings>
```

```
    <system.web>
      <profile>
        <providers>
          <clear/>
          <add name="AspNetSqlProfileProvider"
               connectionStringName="LocalSqlServer"
               applicationName="/"
               type="System.Web.Profile.SqlProfileProvider, System.Web,
               Version=4.0.0.0,
               Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
        </providers>
      </profile>
    </system.web>
  </configuration>
```

和大多数*.config文件相似，不过这个看上去更复杂。基本上我们会使用必要的数据定义一个<connectionString>元素，后面是SqlProfileProvider的具名实例（不管ASPNETDB.mdf的物理路径是什么，这是默认使用的提供程序）。

说明　为了简单，假设只使用位于我们Web应用程序的App_Data子目录中自动生成的ASPNETDB.mdb数据库。

34.10.2　在web.config中定义用户配置

之前提过，用户配置定义在web.config文件中。这个方式的优势是我们可以使用继承的Profile属性以强类型形式和用户配置交互。例如，新建一个叫FunWithProfiles的空网站，添加一个新的*.aspx文件，然后打开web.config文件进行编辑。

我们的目标是创建一个用户配置来模拟会话中用户的地址以及这些用户花在网站上的时间。不足为奇，定义在<profile>元素中的用户配置数据使用一组名称/数据类型对。考虑如下的用户配置，它创建在<system.web>元素区域中：

```
<profile>
  <properties>
   <add name="StreetAddress" type="System.String" />
   <add name="City" type="System.String" />
   <add name="State" type="System.String" />
   <add name="TotalPost" type="System.Int32" />
  </properties>
</profile>
```

在这里，我们为用户配置中的每一个项都指定了名字和CLR数据类型（当然，我们也可以增加邮编、名称等项目，你应该知道怎么做了）。严格地说，type特性是可选的，然而默认值是System.String。就像你期望的那样，可以在用户配置条目中指定很多其他特性，以进一步限制这些信息如何保存在ASPNETDB.mdf中。表34-4展示了一些核心特性。

表34-4 用户配置数据的部分特性

特 性	示 例 值	作 用
allowAnonymous	true | false	限制或允许对值的匿名访问。如果设置为false，匿名用户就不能访问用户配置值
defaultValue	String	如果属性没有被显式设置，则返回这个值
Name	String	这个属性的唯一标识
Provider	String	用于管理这个值的提供程序。它重写web.config或machine.config中的defaultProvider
readOnly	true | false	限制或允许写操作（默认值为false，即不可读）
serializeAs	String | XML | Binary	持久保存到数据库中时值的格式
type	基本类型 | 用户定义的类型	.NET基本类型或类。类名必须完全限定（如MyApp.UserData.ColorPrefs）

在修改当前用户配置时，我们会在实战中研究其中一些特性。但是现在让我们来看如何以编程方式访问这些数据。

34.10.3 以编程方式访问用户配置数据

回忆一下，ASP.NET用户配置API的主要目的是自动化向专门数据库写入（或者从中读取）数据的过程。为了测试，更新我们的Default.aspx文件的UI来增加一组收集街道地址、城市和用户状态的TextBox（和描述性的Label）。同样，增加一个Button类型（命名为btnSubmit）和最后一个Label（叫做lblUserData），它们用来显示持久化的数据，如图34-9所示。

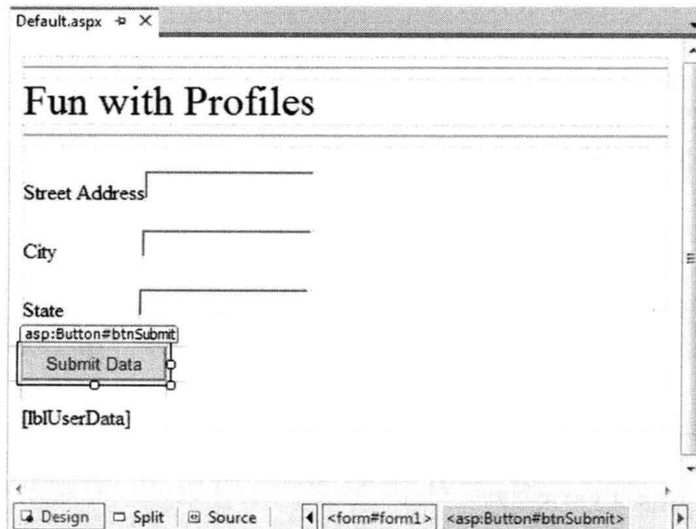


图34-9 FunWithProfiles Default.aspx页面的UI

现在，在按钮的Click事件处理程序中，使用继承的Profile属性来根据用户在相关的TextBox中输入的内容来持久化各种用户配置数据。在ASPNETDB.mdf中每持久化保存一个数据，就从数据库读取一个数据并且格式化成String显示在lblUserData Label类型上。最后，处理页面的Load事件，并且在Label类型上显示相同的信息。这样，当用户访问这个页面的时候，就可以看到它们当前的设置。这里是完整的代码文件：

```
public partial class _Default : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    GetUserAddress();
  }
  protected void btnSubmit_Click(object sender, EventArgs e)
  {
    // 这里发生数据库写入操作
    Profile.StreetAddress = txtStreetAddress.Text;
    Profile.City = txtCity.Text;
    Profile.State = txtState.Text;

    // 从数据库获取设置
    GetUserAddress();
  }

  private void GetUserAddress()
  {
    // 这里发生数据库读取操作
    lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
        Profile.StreetAddress, Profile.City, Profile.State);
  }
}
```

现在运行页面，就会注意到，第一次请求Default.aspx的时候会有一段时间的延迟。原因是ASPNETDB.mdf需要即时创建并放入我们的App_Data文件夹中。（我们也可以通过刷新Solution Explorer窗体并查看App_Data文件夹来自行验证。

我们还会发现，第一次访问这个页面的时候，lblUserData Label没有显示任何用户配置数据，因为我们还没有把数据输入到ASPNETDB.mdf正确的表中。在TextBox控件中输入值并且回发到服务器之后，我们就会发现Label使用持久化的数据进行格式化了。

现在就来看看这个技术最有趣的地方。如果关闭浏览器并重新打开网站的话，就会发现之前输入的用户配置数据确实被持久化了，因为Label显示了正确的信息。问题是，它是怎么记住的呢?

对于这个示例，用户配置API通过我们当前的登录凭证获取并使用了我们的Windows网络标识。然而，如果构建的是公共的网站（用户不属于某个域）的话也没问题，用户配置API整合了ASP.NET表单认证模型并且支持“匿名用户配置”的概念，它允许我们为当前在网站上没有活动标识的用户持久化用户配置数据。

说明　本书不会讲述ASP.NET安全的主题（如基于表单的验证或匿名用户配置）。更多细节请参考.NET Framework 4.5 SDK文档。

34.10.4　分组用户配置数据并且持久化自定义对象

在结束本章之前，再介绍一下用户配置数据是如何定义在web.config文件中的。当前的用户配置只定义了4个数据，并且直接由用户配置类型公开。如果希望构建更复杂的用户配置，可以将相关数据分组到独特的名字之下。考虑如下的更新：

```
<profile>
  <properties>
    <group name ="Address">
      <add name="StreetAddress" type="String" />
      <add name="City" type="String" />
      <add name="State" type="String" />
    </group>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

这次我们定义了一个叫Address的自定义组来公开用户的街道地址、城市和州。要在页面中访问这个数据，就需要更新代码，通过指定Profile.Address来获得每一个子项。例如，这里是更新后的GetUserAddress()方法（Button控件的Click事件处理程序可能需要以相似的方式进行更新）：

```
private void GetUserAddress()
{
  // 这里发生数据库读取操作
  lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
    Profile.Address.StreetAddress,
    Profile.Address.City, Profile.Address.State);
}
```

运行该示例之前，你需要删除App_Data文件夹下的ASPNETDB.mdf，来确保数据库架构被刷新。这样，就可以准确无误地运行这个网站示例了。

说明　一个用户配置可以有任意多个组。只需要在<properties>区域中定义多个<group>元素即可。

最后，值得指出的是，用户配置还可以从ASPNETDB.mdf持久化保存（和获取）自定义对象。例如，假设我们希望构建一个自定义类（或结构）来表示用户的地址数据。用户配置API的唯一需求就是将类型标记为[Serializable]特性，例如：

```
[Serializable]
public class UserAddress
{
  public string Street = string.Empty;
  public string City = string.Empty;
  public string State = string.Empty;
}
```

有了这个类，用户配置定义就可以更新成如下所示（注意这里移除了自定义组，但这不是必需的）：

```
<profile>
  <properties>
    <add name="AddressInfo" type="UserAddress" serializeAs ="Binary"/>
    <add name="TotalPost" type="Integer" />
  </properties>
</profile>
```

如果为用户配置引入[Serializable]类型，type特性就是要持久化类型的完全限定名。从Visual Studio智能感知中可以看到，我们可以选择二进制、XML或字符串数据。既然已经把街道地址信息封装为了自定义类类型，那么就需要再一次更新我们的代码，如下所示：

```
private void GetUserAddress()
{
  // 这里发生数据库读取操作
  lblUserData.Text = String.Format("You live here: {0}, {1}, {2}",
    Profile.AddressInfo.Street, Profile.AddressInfo.City,
    Profile.AddressInfo.State);
}
```

可以肯定的是，用户配置API还有很多内容这里没介绍。例如，Profile属性其实封装了一个叫ProfileCommon的类型。使用这个类型，我们就可以编程方式获取某个用户的所有信息，删除（或增加）用户配置到ASPNETDB.mdf、更新用户配置的某个方面等。

同样，用户配置API有许多可扩展点，可以允许我们优化用户配置管理器如何访问ASPNETDB.mdf数据库的表。正如我们期望的那样，有很多方式可以减少访问数据库的次数。感兴趣的读者可以参考.NET Framework 4.5 SDK文档来了解更多细节。

源代码　FunWithProfiles网站的源代码位于Chapter 34子目录下。

34.11　小结

本章通过研究如何利用HttpApplication类型结束了对ASP.NET知识的学习。你已看到，这个类型提供了大量默认的事件处理程序，它们允许截取各种应用程序级和会话级的事件。本章用大量篇幅讲述几个状态管理技术。其中，视图状态用来在向指定的页面回传间自动重填充页面中HTML组件的值。接下来介绍了应用程序级和会话级数据的区别、cookie管理和ASP.NET应用程序缓存。

本章剩余部分会介绍ASP.NET用户配置API。我们会看到，这个技术提供了现成的方案解决跨会话持久化用户数据的问题。使用网站的web.config文件，我们就可以定义许多配置文件项（包括一组项以及[Serializable]类型），它们会被自动持久化到ASPNETDB.mdf中。

索 引

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

"要想玩转C#，再难找到比这本书更好的资料了！"

——亚马逊读者

".NET与C#百科全书，每个C#程序员必备的经典参考书。"

——I PROGRAMMER

"难得的介绍得深入浅出的语言类图书，既能讲明白各种语言特性，也会解释清楚设计原理及需要解决的问题。"

——豆瓣读者

Pro C# 5.0 and the .NET 4.5 Framework
Sixth Edition

精通C#（第6版）

本书是被誉为"C#圣经"的经典著作，因语言生动流畅、剖析深入、涵盖全面而广受推崇，畅销不衰。曾经获得Referenceware编程图书大奖，并入选Jolt大奖提名。书中探讨了C#语言和.NET平台的各种特性，包括面向对象编程，委托、事件和Lambda表达式的关系，LINQ编程，多线程、并行和异步编程，ADO.NET、WCF、WF、WPF等技术。新版更透彻阐述了C# 5.0和.NET 4.5的新功能。

本书作者为世界级C#专家、C#超级畅销书作家Andrew Troelsen，英文原版一出即成为亚马逊销量最好的C#图书。第5版中文版在豆瓣评分高达9.1分，是众多C#程序员力荐的经典好书。

不论是从零开始的菜鸟，还是小有水平的中级程序员，抑或是已经跻身高手梯队的老码农，都需要用这本书来武装自己，正如一位读者所说，"不藏此书，便不像一名真正的C#程序员"。

Apress®



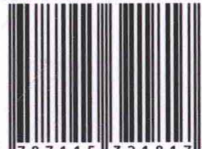
图灵社区：www.ituring.com.cn
新浪微博：@图灵教育　@图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议　计算机/程序设计/C#

人民邮电出版社网址：www.ptpress.com.cn







ISBN 978-7-115-32181-7
定价：159.00元